AIX Version 3 for RISC System/6000™

Communications Programming Concepts

SC23-2206-00

First Edition (March 1990)

This edition of the *AIX Communications Programming Concepts for IBM RISC System/6000* applies to *Version Number 3* of the IBM AIX Base Operating System licensed program, AIX SNA Services/6000, AIX 3278/79 Emulation/6000, and AIX Network Management/6000, and to all subsequent releases of these products until otherwise indicated in new releases or technical newsletters.

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS MANUAL "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

IBM does not warrant that the contents of this publication or the accompanying source code examples, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying source code examples are error-free.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Any reference to an IBM licensed program in this publication is not intended to state or imply that you can use only IBM's licensed program. You can use any functionally equivalent program instead.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas 78758–3493. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright AT&T, 1984, 1985, 1986, 1987, 1988, 1989. All rights reserved.

© Copyright Sun Microsystems, Inc., 1985, 1986, 1987, 1988. All rights reserved.

The Network File System (NFS) was developed by Sun Microsystems, Inc.

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California. We acknowledge the following institutions for their role in its development: the Electrical Engineering and Computer Sciences Department at the Berkeley Campus.

Portion of the code and documentation described in this book were derived from code and documentation developed under the auspices of the Regents of the University of California and have been acquired and modified under the provisions that the following copyright notice and permission notice appear:

© Copyright Regents of the University of California, 1986, 1987. All rights reserved.

Redistribution and use in source and binary forms are permitted provided that this notice is preserved and that due credit is given to the University of California at Berkeley. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission. This software is provided "as is" without express or implied warranty.

This software is derived in part from the ISO Development Environment (ISODE). IBM acknowledges source author Marshall Rose and the following institutions for their role in its development: The Northrup Corporation and The Wollongong Group.

© Copyright Apollo Computer, Inc., 1987. All rights reserved.

© Copyright TITN, Inc., 1984, 1989. All rights reserved.

IBM is a registered trademark of International Business Machines Corporation.

© Copyright International Business Machines Corporation 1987, 1990. All rights reserved.

Trademarks and Acknowledgements

The following trademarks and acknowledgements apply to this book:

AIX is a trademark of International Business Machines Corporation.

AIXwindows is a trademark of International Business Machines Corporation.

Apollo is a trademark of Apollo Computer, Inc.

IBM is a registered trademark of International Business Machines Corporation.

NCK is a trademark of Apollo Computer, Inc.

NCS is a trademark of Apollo Computer, Inc.

Network Computing Kernel is a trademark of Apollo Computer, Inc.

Network Computing System is a trademark of Apollo Computer, Inc.

Network File System and NFS are trademarks of Sun Microsystems, Inc.

RISC System/6000 is a trademark of International Business Machines Corporation.

SNA 3270 is a trademark of International Business Machines Corporation.

UNIX was developed and licensed by AT&T and is a registered trademark of AT&T Corporation.

About This Book

This book contains conceptual and procedural information about various communications programming tools.

Who Should Use This Book

This book is intended for programmers who know the C language and have some knowledge of communications applications and who want to create and implement communications programs.

How to Use This Book

The chapters of this book contain concepts, procedures and examples for programs for communications on systems and networks. The concept sections are divided into overviews and subsequent information. The procedures and examples follow the conceptual information.

Overview of Contents

This book contains chapters on the following topics:

- The Generic Data Link Control (GDLC) is a generic interface definition that allows both application and kernel users to have a common set of commands to control DLC device managers within the AIX Version 3 system. The GDLC interface specifies requirements for entry point definitions, functions provided, and data structures for all DLC device managers. This section contains information on GDLC criteria, implementing GDLC interfaces, installing data link controls, solving DLC problems, and programming for DLC.
- The eXternal Data Representation (XDR) is a standard for the description and encoding of data. XDR uses a language to describe data formats, but the language is used only for describing data and is not a programming language. This section contains information on XDR criteria, implementing XDR, installing XDR, solving XDR problems, and programming for XDR.
- The AIX 3270 Host Connection Program/6000 Licensed Program (HCON) application programming interface (API) allows AIX users to develop applications programs that communicate with a host system. This chapter/section contains information on HCON data structures, the file transfer programming interface and the HCON application program interface. Also included is information on how to install the HCON API on a host system, how to incorporate automatic logon in your HCON applications, and error and status information for program troubleshooting.
- The Network Computing System (NCS) enables the distribution of processing tasks across resources in a network or internet by maintaining databases that control the information about the resources. NCS consists of three components: the Remote Procedure Call runtime library, the Location Broker, and the Network Interface Definition Language compiler. This chapter provides detailed information on the working of NCS and its components. In addition, it provides brief introductions to the various library routines used in NCS.

- The AIX Network Management/6000 Licensed Program (**xgmon**) is a network management program for monitoring TCP/IP networks. It assists in monitoring the status of all the machines on a network by communicating with SNMP agents and receiving SNMP–based traps. This chapter provides detailed information on the **xgmon** programming utility, which enables you to extend the **xgmon** program, as well as information on the SNMP API Subroutine Library, which allows you to create SNMP manager applications. Also included is information on the Simple Network Management Protocol (SNMP), the SNMP daemon, and the SNMP Command Line Manager.
- The Remote Procedure Call (RPC) is a protocol that provides the high-level communications paradigm used in the operating system. RPC implements a logical client-to-server communications system designed specifically for the support of network applications. This section contains information on the messages, authentication, language, and protocol compiler for RPC.
- Systems Network Architecture (SNA) is a specification that formally defines the functional responsibilities for components of a data communications system and specifies how those components must interact. This chapter contains information to help you to set up and customize SNA for your system. It includes information on SNA programming concepts.
- The Sockets facility is a Berkeley Software Distribution (BSD) programming interface that provides applications programs with interprocess and network I/O communication capabilities. The Sockets mechanism consists of socket subroutines which enable local or remote application programs to set up virtual connections and exchange data. In AIX Version 3.1, the Sockets facility serves as the application program interface for TCP/IP. This section/chapter provides information on the Sockets facility and its components. It contains information about socket creation, connection, and use in application programs. In addition, this section/chapter provides brief descriptions of socket data structures, the socket Kernel Service Subroutines, and Network Library Subroutines.
- The X.25 is an international standard protocol that allows intercommunication between systems. It is particularly useful for communicating with people using different computer systems and for applications that access public data bases.

Highlighting

The following highlighting conventions are used in this book:

Bold Identifies commands, keywords, files, directories, and other items whose names are predefined by the system.

Italics Identifies parameters whose actual names or values are to be supplied by the user.

`Monospace` Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

Related Publications

The following book contains information about or related to communications:

- *AIX Calls and Subroutines Reference for IBM RISC System/6000*, Order Number SC23–2198.
- *AIX Communications Concepts and Procedures for IBM RISC System/6000*, Order Number SC23–2203.
- *AIX Files Reference for IBM RISC System/6000*, Order Number SC23–2200.

- *AIX General Programming Concepts for IBM RISC System/6000*, Order Number SC23–2205.
- *AIX Kernel Extensions and Device Support Programming Concepts for IBM RISC System/6000*, Order Number SC23–2207.

Ordering Additional Copies of This Book

To order additional copies of this book, use Order Number SC23–2206.

Table of Contents

Chapter 1. Generic Data Link Control Environment 1–1
Implementing the GDLC Interface 1–3
Installing Data Link Controls 1–4
List of the DLC Interface Entry Points 1–4
Understanding DLC Interface ioctl Entry Point Operations 1–5
Using Special Kernel Services 1–7
Understanding DLC Problem Determination 1–8
Data Link Control (DLC) Reference Information 1–14
Token–Ring Data Link Control 1–16
DLCTOKEN Device Manager Functions 1–17
Protocol Support 1–18
Name-Discovery Service 1–19
Direct Network Services 1–20
Connection Contention 1–20
Link Session Initiation 1–20
Link Session Termination 1–21
DLCTOKEN Programming Interfaces 1–21
IEEE 802.3 Ethernet Data Link Control 1–26
DLC8023 Device Manager Functions 1–27
Protocol Support 1–28
Name Discovery Services 1–29
Direct Network Services 1–30
Connection Contention 1–30
Link Session Initiation 1–30
Link Session Termination 1–30
DLC8023 Programming Interfaces 1–31
Standard Ethernet Data Link Control 1–35
DLCETHER Device Manager Functions 1–36
Protocol Support 1–37
Name-Discovery Services 1–38
Direct Network Services 1–39
Connection Contention 1–39
Link Session Initiation 1–39
Link Session Termination 1–40
DLCETHER Programming Interfaces 1–40
Synchronous Data Link Control 1–44
DLCSDLC Device Manager Functions 1–44
DLCSDLC Protocol Support 1–44
DLCSDLC Programming Interfaces 1–47
Qualified Logical Link Control 1–52
DLCQLLC Device Manager Functions 1–52
DLCQLLC Programming Interfaces 1–53

Chapter 2. Lists of DBM, NDBM, and NIS Subroutines 2–1
Alphabetical List of DBM Subroutines 2–1
Alphabetical List of NDBM Subroutines 2–2
Alphabetical List of NIS Subroutines 2–3

Chapter 3. eXternal Data Representation (XDR) 3–1
A Canonical Standard 3–1
Basic Block Size 3–2
Planned Enhancements 3–2
Understanding the XDR Subroutine Format 3–3
Using the XDR Library 3–4
XDR with RPC 3–4
XDR Operation Directions 3–4
Understanding the XDR Language Specification 3–5
Lexical Notes 3–5
Declarations, Enumerations, Structures, and Unions 3–5
Syntax Notes 3–7
Understanding XDR Data Types 3–8
Integer Data Types 3–8
Enumeration Data Type 3–9
Boolean Data Type 3–9
Floating-Point Data Type 3–9
Opaque Data Type 3–11
Array Data Type 3–12
Strings 3–13
Structures 3–13
Discriminated Unions 3–14
Voids 3–15
Constants 3–15
Type Definitions 3–15
Optional Data 3–16
Understanding XDR Library Filter Primitives 3–17
Using XDR Basic Filter Primitives 3–17
Using XDR Constructed Filter Primitives 3–18
Understanding XDR Non–Filter Primitives 3–20
Creating and Using XDR Data Streams 3–20
Manipulating an XDR Data Stream 3–21
Implementing an XDR Data Stream 3–21
Destroying an XDR Data Stream 3–22
Alphabetical List of XDR Subroutines and Macros 3–24
Functional List of XDR Subroutines and Macros 3–26
Using the XDR Library Filter Primitives 3–26
Using the XDR Library Non–Filter Primitives 3–27
List of XDR Examples 3–28
Example Passing Linked Lists Using XDR 3–29
Example Showing the Justification for Using XDR 3–32
Example Showing the Use of Pointers in XDR 3–35
Example Using an XDR 3–36
Example Using an XDR Array 3–37
Example Using an XDR Data Description 3–40
Example Using an XDR Discriminated Union 3–42

Chapter 4. 3270 Host Connection Program/6000 (HCON) 4–1
Understanding the File Transfer Program Interface 4–4
Understanding the HCON Application Program Interface (API) 4–8
Understanding the AIX Interface for HCON API 4–11
Understanding the Host Interface for HCON API 4–15
Host Session Control 4–15
Host Message Interface 4–16
Host Interface Errors 4–16
Understanding Explicit and Implicit Logon 4–17
Understanding AUTOLOG 4–19
Understanding the Logon Assist Feature (LAF) 4–20
Understanding the Automatic Logon Commands 4–25
Understanding HCON Programming Examples 4–25
File Transfer Programming Examples 4–25
API Programming Examples 4–26
Running Example Programs in TSO 4–26
AUTOLOG and LAF Program Examples 4–26
Compiling Programs 4–27
File Transfer Program Interface Error Codes 4–28
HCON AIX API Error Codes 4–42
Implementation Specifics 4–49
HCON Host API Errors 4–50
Host API System Errors – VM/CMS 4–51
Host API System Errors – MVS/TSO 4–52
cname Example VM/CMS File Transfer Program 4–53
fname Example FORTRAN File Transfer Program 4–55
pname Example Pascal File Transfer Program 4–56
g32_sampl Example Program 4–58
g32_test Example Program 4–61
g32_3270 Example Program 4–62
How To Install the HCON MVS/TSO Host API 4–64
How To Install the HCON VM/CMS Host API 4–66
How To Compile a File Transfer Program 4–68
How To Compile an AIX API Program 4–69
How To Compile a Host HCON API Program 4–70
How To Use a Logon Assist Feature Script 4–71
How to Use the Sample LAF Scripts 4–71
Testing a LAF Script 4–72
How To Use an AUTOLOG Profile 4–74
Using an AUTOLOG Profile 4–74
Testing the AUTOLOG Profile 4–76
Linking to AUTOLOG 4–77

Chapter 5. Network Computing System (NCS) **5–1**
Understanding NCS 5–2
The Remote Procedure Call (RPC) Runtime Library (NCS) 5–13
Routines 5–13
Client Routines 5–14
Server Routines 5–14
Conversion Routines 5–15
Interface Definitions and the NIDL Compiler (NCS) 5–16
The Banking Example 5–17
The binop Example 5–18
Stub Functionality 5–20
Marshalling and Conversion 5–20
Handles and Binding 5–21
Client Switches 5–24
Writing Programs That Use the Network Computing System 5–24
Managing Handles and Bindings 5–34
Writing the Client Program 5–43
Writing the Server Program 5–46
Writing the Manager Procedures 5–48
Building an Application 5–49
Using C Syntax with NIDL 5–50
Using Pascal Syntax with NIDL 5–62
Using NCS with FORTRAN Programs 5–73
The Location Broker (NCS) 5–75
Understanding the Location Broker 5–75
NCS Daemons and Utilities 5–80
rpc_$ Library Routines (NCS) 5–81
Constants 5–81
External Variable 5–81
Data Types 5–81
rpc_$ Status Codes 5–83
List of rpc_$ Library Routines 5–85
pfm_$ Library Routines (NCS) 5–86
Cleanup Handlers 5–86
Constant 5–86
Data Types 5–86
pfm_$ Status Codes 5–87
List of pfm_$ Library Routines 5–88
lb_$ Library Routines (NCS) 5–89
Constants 5–89
External Variable 5–89
Data Types 5–90
lb_$ Status Codes 5–92
List of lb_$ Library Routines 5–93
uuid_$ Library Routines (NCS) 5–94
External Variables 5–94
Data Types 5–94
List of uuid_$ Library Routines 5–96
Glossary 5–97

Chapter 6. Network Management/6000 (xgmon) **6–1**
Understanding the Simple Network Management Protocol (SNMP) 6–3
Understanding the Management Information Base (MIB) 6–3
Understanding Terminology Related to MIB Variables 6–6
Using the Management Information Base (MIB) Database 6–6
Understanding How a Monitor Functions 6–8
Understanding How an Agent Functions 6–8
Working with Management Information Base (MIB) Variables 6–9
Using the SNMP API Subroutine Library 6–10
Alphabetic List of API Subroutines 6–12
Understanding the SNMP Daemon 6–13
Configuring the SNMP Daemon 6–13
Understanding SNMP Daemon Processing 6–14
Understanding SNMP Daemon Support for the EGP Family of MIB Variables . 6–16
Understanding SNMP Daemon Support for SET Request Processing 6–17
Understanding SNMP Daemon RFC Conformance 6–20
Understanding SNMP Daemon Implementation Restrictions 6–21
Understanding the SNMP Command Line Manager 6–22
Understanding the xgmon Programming Utility 6–23
Extending xgmon Intrinsic Functions 6–23
Creating xgmon Library Commands 6–24
Programming Virtual G Machines (VGMs) 6–24
Understanding the Internal Database 6–25
Formatting the Virtual G Machine (VGM) Windows 6–25
Working with Virtual G Machine (VGM) Variables 6–26
Working with Virtual G Machine (VGM) Data Types 6–29
Understanding the Virtual G Machine (VGM) Run-Time Environment 6–30
Understanding the Structure of xgmon Library Programs 6–30
Using Simple Statements 6–32
Using Iteration and Conditional Statements 6–33
Using Expressions 6–34
Using Operators 6–34
Using Intrinsic Functions 6–35
Alphabetic List of Intrinsic Functions 6–37
Functional List of Intrinsic Functions 6–39
Database Operations 6–39
Host Information 6–40
String Manipulation 6–40
Formatted Output 6–40
File I/O 6–40
Virtual G Machine Control 6–41
Graphics Functions 6–41
How to Create xgmon Intrinsic Functions 6–43
How to Create xgmon Library Commands 6–46
How to Modify Existing xgmon Library Commands 6–48

Chapter 7. Remote Procedure Call (RPC) 7–1
Understanding the RPC Model 7–2
Transports and Semantics 7–3
RPC in the Binding Process 7–4
Understanding the RPC Message Protocol 7–5
Understanding the RPC Protocol Requirements 7–5
Understanding the RPC Messages 7–5
Understanding an RPC Call Message 7–6
Understanding an RPC Reply Message 7–7
Marking Records in RPC Messages 7–9
Understanding RPC Authentication 7–10
Understanding RPC Authentication Protocol 7–10
Understanding NULL Authentication 7–11
Understanding UNIX Authentication 7–11
Understanding Data Encryption Standard (DES) Authentication 7–12
Understanding Data Encryption Standard (DES) Authentication Protocol 7–14
Understanding Diffie-Hellman Encryption 7–15
Understanding the RPC Port Mapper Program 7–17
Registering Ports 7–17
Understanding Port Mapper Protocol 7–18
Understanding Port Mapper Procedures 7–19
Programming in RPC 7–20
Assigning Program Numbers 7–20
Assigning Version Numbers 7–21
Assigning Procedure Numbers 7–21
Using Registered RPC Programs 7–21
Using the Intermediate Layer of RPC 7–23
Allocating Memory with XDR 7–26
Starting RPC from the inetd Daemon 7–27
Compiling and Linking RPC Programs 7–27
Understanding the RPC Features 7–29
Batching Remote Procedure Calls 7–29
Broadcasting Remote Procedure Calls 7–30
Understanding RPC Call-back Procedures 7–30
Understanding the select Subroutine on the Server Side 7–30
Understanding the RPC Language 7–31
Understanding RPC Language Descriptions 7–31
Definitions 7–31
Structures 7–32
Unions 7–32
Enumerations 7–33
Type Definitions 7–33
Constants 7–34
Programs 7–34
Declarations 7–35
RPCL Syntax Requirements for Program Definition 7–36
Exceptions to the RPCL Rules 7–36
Understanding the rpcgen Protocol Compiler 7–37
Converting Local Procedures into Remote Procedures 7–38
Generating XDR Routines 7–38
Understanding the C Preprocessor 7–38

Changing Time Outs 7–39
Handling Broadcast on the Server Side 7–39
Other Information Passed to Server Procedures 7–39
Alphabetical List of RPC Subroutines and Macros 7–41
Functional List of RPC Subroutines and Macros 7–44
List of RPC Examples 7–49
Example Using UNIX Authentication 7–50
Example Using DES Authentication 7–53
Example of an RPC Language ping Program 7–56
Example of Broadcasting a Remote Procedure Call 7–57
Example Using the Highest Layer of RPC 7–58
Example Using the Intermediate Layer of RPC 7–59
Example Showing How RPC Passes Arbitrary Data Types 7–61
Example Using the Lowest Layer of RPC 7–63
The Lowest Layer of RPC from the Client Side 7–65
Example Using the select Subroutine 7–68
Example Using rcp on TCP 7–69
Example Using Multiple Program Versions 7–73
Example Converting Local Procedures into Remote Procedures 7–75
Example Generating XDR Routines 7–80
Example Using RPC Callback Procedures 7–84

Chapter 8. AIX SNA Services/6000 8–1
AIX SNA Services/6000 Subroutines 8–1
AIX SNA Services/6000 Operating System Subroutines 8–1
AIX SNA Services/6000 Operating System Subroutine Interfaces 8–1
AIX SNA Services/6000 Library Subroutines 8–2
AIX SNA Services/6000 Special Files 8–5
luxsna.h Include File 8–5
allo_str Structure 8–6
alloc_listen Structure 8–9
attr_str Structure 8–10
confirm_str Structure 8–11
cp_str Structure 8–11
deal_str Structure 8–13
erro_str Structure 8–14
ext_io_str Structure 8–15
flush_str Structure 8–20
fmh_str Structure 8–21
get_parms Structure 8–22
gstat_str Structure 8–22
pip_str Structure 8–24
prep_str Structure 8–24
read_out Structure 8–25
stat_str Structure 8–27
write_out Structure 8–27
Constant Definitions 8–28
Request Code Constants 8–30
Developing Special SNA Functions 8–32
Document Guide 8–32
Configurations 8–34

Functional Characteristics 8–34
AIX SNA Services/6000 Terminology 8–36
IBM AIX SNA Services/6000 LU0 Facility 8–37
Document Guide 8–37
Using the Menus 8–38
Defining LU0 Secondary Support 8–38
Defining LU0 Primary Support 8–40
Defining LU0 Primary Application Logical Units 8–41
Application Program Interface 8–43
Applications 8–43
Writing Transaction Programs for AIX SNA Services/6000 8–44
Guidelines for Writing Transaction Programs 8–44
AIX SNA Services/6000 Example Programs 8–45
Local Transaction Program Example Program 8–46
Remote Transaction Program Example Program 8–48
Mapped Local Transaction Program Example Program 8–51
Mapped Remote Transaction Program Example Program 8–52
Transferring Files Using AIX SNA Services/6000 8–55
Sending Files from a Local AIX Node to a Remote AIX Node 8–56
Receiving Files from a Remote AIX Node 8–56
Writing Generic AIX SNA Services/6000 Programs 8–57
Generic AIX SNA Services/6000 Example Program 8–57

Chapter 9. Sockets **9–1**
Critical Attributes 9–1
Sockets Background 9–1
Sockets Facilities 9–2
Understanding the Sockets Interface 9–3
Sockets Interface 9–3
Socket Interface to Network Facilities 9–4
Understanding Socket Subroutines 9–5
Socket Subroutines 9–5
Understanding Socket Header Files 9–5
Socket Header Files 9–5
Understanding Socket Communications Domains 9–6
Understanding Socket Addresses 9–8
Socket Address Storage 9–8
Socket Addresses in TCP/IP 9–9
Understanding Socket Types and Protocols 9–9
Socket Types 9–10
Socket Protocols 9–11
Understanding Socket Creation 9–12
Socket Creation 9–12
Binding Names to Sockets 9–13
Binding Addresses to Sockets 9–13
Obtaining Socket Addresses 9–14
Understanding Socket Connections 9–15
Socket Connection 9–15
Server Connections 9–16
Connectionless Datagram Services 9–16
Understanding Socket Options 9–17

Socket Options 9–17
Understanding Socket Data Transfer 9–18
Socket Data Transfer 9–18
Out–of–Band Data 9–19
Socket I/O Modes 9–20
Understanding Socket Shutdown 9–21
Socket Shutdown 9–21
Closing Sockets 9–21
Understanding Network Address Translation 9–22
Network Address Translation 9–22
Understanding Domain Name Resolution 9–24
Domain Name Resolution 9–24
Understanding Socket Examples 9–26
Socket Examples 9–26
List of Socket Kernel Service Subroutines 9–27
List of Network Library Socket Subroutines 9–27
List of Socket Header Files 9–29
List of Socket Examples 9–29
Socketpair Communication Example Program 9–30
Reading UNIX Datagrams Example Program 9–31
Sending UNIX Datagrams Example Program 9–32
Reading Internet Datagrams Example Program 9–33
Sending Internet Datagrams Example Program 9–34
Initiating Internet Stream Connections Example Program 9–35
Accepting Internet Stream Connections Example Program 9–36
Initiating UNIX Stream Connections Example Program 9–38
Accepting UNIX Stream Connections Example Program 9–39
Checking for Pending Connections Example Program 9–41

Chapter 10. X.25 Communications 10–1
X.25 Overview 10–1
The X.25 Application Programming Interface (API) 10–1
Using the X.25 Subroutines 10–2
Using X.25 Applications Written for Previous Releases 10–3
Using the X.25 Structures and Flags 10–5
Understanding X.25 Error Codes 10–6
Using Processes in X.25 Applications 10–6
Providing Security in X.25 Applications 10–6
X.25 Calls: API Level 10–8
X.25 API: Initializing and Terminating 10–8
/dev/x25sn Special File 10–9
X.25 API: Using the Connection Identifier for Calls 10–9
X.25 API: Using Counters to Correlate Messages 10–10
X.25 API: Listening for Incoming Calls 10–12
X.25 API: Making and Receiving a Call 10–13
X.25 API: Transferring and Acknowledging Data 10–14
X.25 API: Clearing, Resetting, and Interrupting Calls 10–15
List of X.25 API Error Codes 10–18
List of System Error Codes 10–20
X.25 Example Programs Overview 10–21
Preparing, Compiling, and Running the Example Programs 10–21

Using the Example Code 10–21
X.25 Example Program pvcrcv: Program Description 10–22
Do until the end–of–transmission indicator is received: 10–22
When the end–of–transmission indicator has been received: 10–22
X.25 Example Program pvcrcv: Receive Data Using a PVC 10–23
X.25 Example Program pvcxmit: Program Description 10–27
X.25 Example Program pvcxmit: Send Data Using a PVC 10–28
Example Program pvcxmit 10–28
X.25 Example Program svcrcv: Program Description 10–32
Repeat until a clear-indication message arrives: 10–32
X.25 Example Program svcrcv: Receive a Call Using an SVC 10–33
X.25 Example Program svcxmit: Program Description 10–38
X.25 Example Program svcxmit: Make a Call Using an SVC 10–39

Index **X–1**

Chapter 1. Generic Data Link Control Environment

The Generic Data Link Control (GDLC) is a generic interface definition that allows both application and kernel users to have a common set of commands to control DLC device managers within the AIX Version 3 system. The GDLC interface specifies requirements for entry point definitions, functions provided, and data structures for all DLC device managers. This section contains information on GDLC criteria, implementing GDLC interfaces, installing data link controls, solving DLC problems, and programming for DLC.

Generic Data Link Control Environment Overview

The Generic Data Link Control (GDLC) is a generic interface definition that allows both application and kernel users to have a common set of commands to control DLC device managers within the AIX Version 3 system. The GDLC interface specifies requirements for entry point definitions, functions provided, and data structures for all DLC device managers. An example set of DLCs that conform to the GDLC interface are:

- DLCTOKEN (Token-Ring)
- DLCETHER (Standard Ethernet)
- DLC8023 (IEEE 802.3 for Ethernet)
- DLCSDLC (Synchronous Data Link Control)
- DLCQLLC (Qualified Logical Link Control).

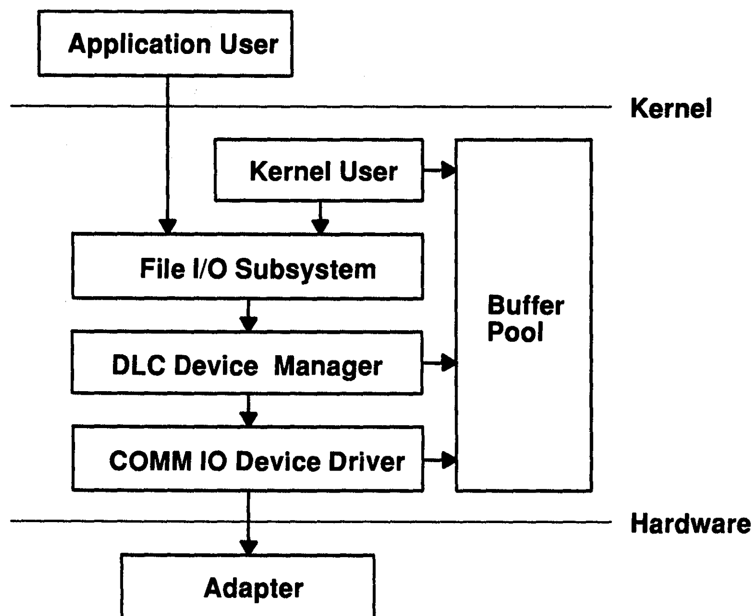
DLC device managers perform higher layer protocols and functions beyond the scope of a kernel device driver, but still reside within the kernel for maximum performance and use a kernel device driver for their I/O requests to the adapter. A user of a DLC can be located above or within the kernel.

An example of a DLC device manager is Synchronous Data Link Control (SDLC) or IEEE 802.2 Data Link Control. Each DLC device manager operates with a specific device driver or set of device drivers. SDLC, for example, operates with the IBM Multiprotocol device driver for the GL product and its associated adapter.

The basic structure of a DLC environment is shown in the following figure. Users within the kernel have access to the Communications memory buffers (**mbufs**) and call the **dd** entry points by way of the **fp** kernel services. Users above the kernel use the standard interface-to-kernel device drivers, and the file system then calls the **dd** entry points. Data transfers in this case require a move of the data between user and kernel space.

See DLC's environment and its relationship to other components in the following figure:

DLC Device Manager Environment

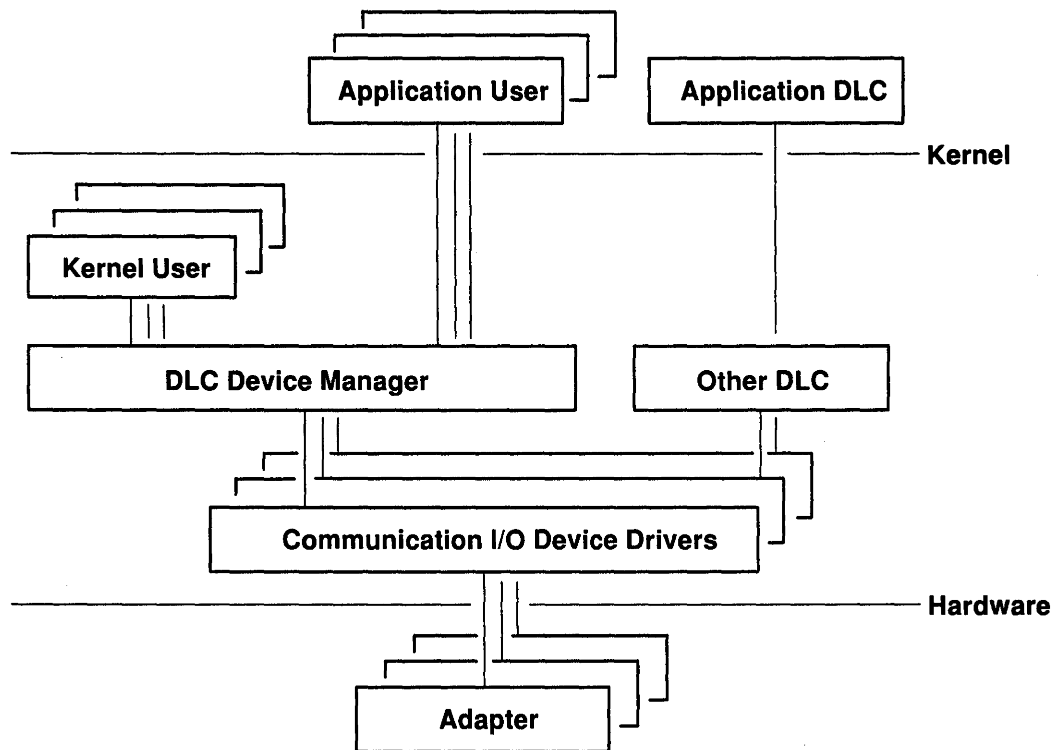


Application User	Resides above the kernel as an application or access method.
Kernel User	Resides within the kernel as a kernel process or device manager.
File I/O Subsystem	Converts the file descriptor and file pointer subroutines to file pointer accesses of the switch table.
Buffer Pool	Provides data buffer services for the communications subsystem.
Comm I/O Device Driver	Controls hardware adapter I/O and DMA registers, and routes receive packets to multiple DLCs.
Adapter	Attaches to the communications media.

A device manager written in accordance with GDLC specifications can run on all AIX Version 3 hardware configurations that contain a communications device driver and its target adapter. Each of these device managers can support multiple users above and multiple device drivers and adapters below. In general, users can operate concurrently over a single adapter, or each user can operate over multiple adapters. Some DLC device managers may vary depending on their particular protocol constraints.

The following figure illustrates a multiple user configuration:

Example of Multiple-user, Multiple-Adapter Configuration



Meeting the GDLC Criteria

There are several criteria that must be met in order for a GDLC interface to be a truly generic interface. This interface must do the following:

- Be flexible and be accessible to both application and kernel users.
- Have multi-user and multi-adapter capability for protocols that can take advantage of multiple sessions and ports.
- Support both connection-oriented and connectionless types of DLC device managers.
- Provide a means to pass data in a transparent mode for those users that have special requirements beyond the scope of the DLC device manager in use.

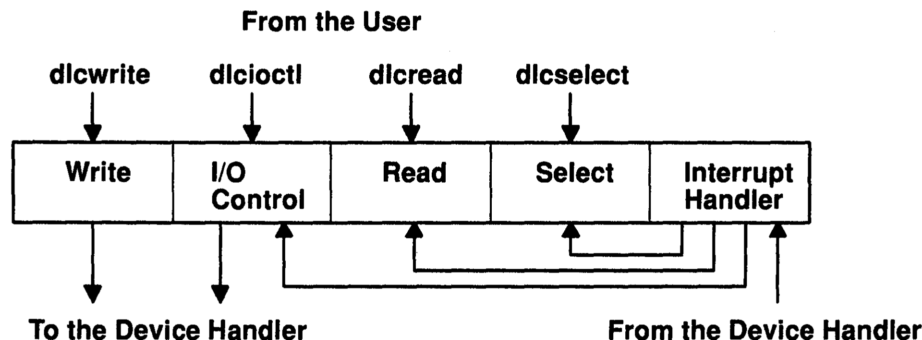
Implementing the GDLC Interface

Each DLC device manager is a standard /**dev** entry that operates in the kernel as a multiplexed device manager for a particular protocol. Each **open** subroutine to a DLC device manager for an adapter port that is not already in use by DLC causes a kernel process to be created and an **open** subroutine to be issued to the target adapter's device handler. Additional **open** subroutines can be issued to the same DLC device manager in order to talk to multiple adapter ports of the same protocol. Any **Open** subroutines that target the same port do not create additional kernel processes, but rather link the new **open** subroutine with the existing process. There is always one kernel process for each port in use.

The internal structure of a DLC device manager has the same basic structure as a kernel device handler, except that the interrupt handler is replaced by a kernel process for handling

asynchronous events. The read, write, I/O control, and select blocks function exactly the same, as shown in the following figure:

Standard Kernel Device Manager



Installing Data Link Controls

Each DLC can be installed separately or in a group. Once the DLCs are installed, you must add the DLC device to the system in order to make it useable. You may want to display or list information about an installed DLC to verify that installation and the addition of an installed DLC was successful. Listing current DLC attributes is useful when one must check to see if any changes are necessary.

On heavily used systems, fine tuning or changing may be necessary. If receive performance is sluggish and the system error log indicates that the DLC is experiencing ring queue overflows between the DLC and its device handler, the operator can change the DLC's queue depth for incoming data. Finally, you may want to remove an installed DLC from the system when changing networks, such as switching from the SDLC device manager to the Token-Ring device manager. All of these DLC procedures can be accomplished through the System Management Interface Tool (SMIT) install menus or through the command line.

List of the DLC Interface Entry Points

dlcclose Closes a GDLC channel.

dlcconfig Configures the GDLC device manager.

dlcioctl Issues specific commands to the GDLC.

dlcmpx Decodes the device handler's special file name that was appended to the **open** subroutine.

dlcopen Opens a GDLC channel.

dlcread Reads receive data from the GDLC.

dlcselect Selects asynchronous criteria from the GDLC such as receive data completion and exception conditions.

dlcwrite Writes transmit data to the GDLC.

Understanding DLC Interface ioctl Entry Point Operations

The GDLC interface supports the following **ioctl** subroutine operations:

DLC_ENABLE_SAP Enables a service access point (SAP).

DLC_DISABLE_SAP Disables a SAP.

DLC_START_LS Starts a link station on a particular SAP as caller or listener.

DLC_HALT_LS Halts a link station.

DLC_TRACE Traces a link station's activity for short or long activities.

DLC_CONTACT Contacts a remote station for a particular local link station.

DLC_TEST Tests the link to a remote for a particular local link station.

DLC_ALTER Alters a link station's configuration parameters.

DLC_QUERY_SAP Queries statistics of a particular SAP.

DLC_QUERY_LS Queries statistics of a particular link station.

DLC_ENTER_LBUSY Enters local busy mode on a particular link station.

DLC_EXIT_LBUSY Exits local busy mode on a particular link station.

DLC_ENTER_SHOLD Enters short hold mode on a particular link station.

DLC_EXIT_SHOLD Exits short hold mode on a particular link station.

DLC_GET_EXCEP Returns asynchronous exception notifications to the application user.

Note: This **ioctl** subroutine operation is not used by the kernel user since all exception conditions are passed to the kernel user by way of their exception handler.

IOCINFO Returns a structure that describes the GDLC device manager. See the **sys/devinfo.h** file.

Service Access Point (SAP)

A service access point (SAP) identifies a particular user service that will send and receive a specific class of data. This allows different classes of data to be routed separately to their corresponding service handlers. Those DLCs that support multiple concurrent SAPs have SAP addresses known as Destination SAP and Source SAP imbedded in their packet headers. DLCs that can only support a single SAP do not need or use SAP addressing, but still have the concept of enabling the one SAP. In general, there is a SAP enabled for each DLC user on each port.

Most SAP address values are defined by IEEE-standardized network management entities. Some of the common SAP addresses are:

Null SAP (x'00')
: Provides some ability to respond to remote nodes even when no SAP has been enabled. This SAP supports only connectionless service and responds only to XID and TEST Link Protocol Data Unit (LPDUs).

SNA Path Control (x'04')
: Denotes the default individual SAP address used by SNA nodes.

PC Network NETBIOS (x'F0')
: Used for all DLC communication that is driven by NETBIOS emulation.

Discovery SAP (x'FC')
: Used by IBM LAN name discovery services.

Global SAP (x'FF')
: Identifies all active SAPs.

Link Station (LS)

A link station (LS) identifies an attachment between two nodes for a particular SAP pair. This attachment can operate as a connectionless service (datagram) or connection-oriented service (fully sequenced data transfer with error recovery). In general, there is one LS started for each remote attachment.

Local Busy Mode

Whenever an LS is operating in a connection-oriented mode (contacted) and wishes to stop the remote station's sending of information packets for reasons such as resource outage, notification can be sent to the remote station to cause the local station to enter local busy mode. Once resources are available, the local station will notify the remote that it is no longer busy and that information packets can flow again. Only sequenced information packets are stopped with local busy mode. All other types of data are unaffected.

Short Hold Mode

Use the short hold mode of operation when operating over certain data networks that have the following characteristics:

- Short call setup time
- Tariff structure that specifies a relatively small fee for the call setup compared to the charge for connect time.

With short hold mode an attachment between two stations is maintained only while there is data available for transfer between the two stations. When there is no data to send, the attachment is cleared after a certain time out and established again when there is new data to transfer.

Testing and Tracing a Link

To test an attachment between two stations an LS can be instructed to send a test packet from the local station. This packet is echoed back from the remote station if the attachment is operating correctly.

Some data links may be limited in their support of this function due to protocol constraints. SDLC, for example, only generates the test packet from the host or primary station. Most other protocols, however, allow test packets to be initiated from either station.

To trace a link, line data and special events (such as station activation, termination, and time outs) can be logged in the system's generic trace facility for each link station (LS). This function helps determine the cause of certain communications attachment problems. The GDLC user may select either short or long entries to be traced. Short entries consist of up to 80 bytes of line data, while long entries allow full packets of data to be traced.

Note: Tracing can be activated when an LS is started, or it may be activated or terminated dynamically anytime after an LS has been started.

Statistics

Both SAP and LS statistics can be queried by a GDLC user. The statistics for a SAP consist of the current SAP state and information about the device handler. LS statistics consist of the current station states and various Reliability/Availability/Serviceability counters that have monitored the activity of the station since it was started.

Using Special Kernel Services

GDLC provides special services for a kernel user, with the understanding that a trusted environment must exist within the kernel. Instead of the DLC device manager copying asynchronous event data into user space, the kernel user must specify function pointers to special routines called function handlers. The function handlers are called by the DLC at the time of execution. This allows maximum performance between the kernel user and the DLC layers. Each kernel user is required to keep the number of function handlers down to a minimum path length, and to use the communications memory buffer (**mbuf**) scheme.

A function handler must never call another DLC entry directly, since the call would be made under lock, causing a fatal sleep. The only exception to this general rule is that a kernel user is allowed to call the **dlcwritex** entry during its service of any of the four receive data functions. Calling the **dlcwritex** entry allows immediate responses to be generated without an intermediate task switch.

Special logic is required within the DLC device manager to check the process identification of the user calling a write. If it is a DLC process and the internal queueing capability of the DLC has been exceeded, the write is sent back with a bad return code (EAGAIN return value) instead of putting the calling process (DLC) to sleep. It is then up to the calling user subroutine to return a special notification to DLC from its receive data function to ensure a retry of the receive buffer at a later time.

The following routines are user provided function handlers:

Datagram Data Received
Called any time a datagram packet is received for the kernel user.

Exception Condition
Called any time an asynchronous event occurs that must notify the kernel user, such as SAP Closed or Station Contacted.

I-Frame Data Received
Called each time a normal sequenced data packet is received for the kernel user.

Network Data Received
Called any time network-specific data is received for the kernel user.

XID Data Received
Called any time an exchange identification (XID) packet is received for the kernel user.

The **dlcread** and **dlcselect** entry points for DLC are not called by the kernel user because the asynchronous functional entries are called by the DLC device manager directly. Generally, any queuing of these events must occur in the user's function handler. If, however, the kernel user cannot handle a particular receive packet, the DLC device manager may hold the last receive buffer and enter one of two special user busy modes.

User Terminated Busy Mode (I-frame only)

If the kernel user cannot handle a received I-frame (due to problems such as queue blockage), a –1 return code is given back and DLC holds the buffer pointer and enters local busy mode to stop the remote station's I-frame transmissions. The kernel user must call the Exit Local Busy function to reset local busy mode and start the reception of I-frames again. Only normal sequenced I-frames can be stopped. XID, datagram, and network data are not affected by local busy mode.

Timer Terminated Busy Mode (all frame types)

If the kernel user cannot handle a particular receive packet, and wants DLC to hold the receive buffer for a short period and then re-call the user's receive function, a –2 return code is given back to DLC. If the receive packet is a sequenced I-frame, the station enters local busy mode for that period. In all cases, a timer is started, and once the timer expires, the receive data functional entry is called again.

Understanding DLC Problem Determination

Each of the generic data link controls provide problem determination data that can be used to isolate network problems. Three types of diagnostic information are provided:

- Status
- Error Log
- Link Trace.

Understanding DLC Status Information

Status can be obtained for a service access point (SAP) or a link station by issuing DLC_QUERY_SAP and DLC_QUERY_LS **ioctl** subroutines to the specific DLC kernel device manager in use.

Individual device driver statistics can be obtained with the DLC_QUERY_SAP **ioctl** subroutine from various devices such as:

- Token-Ring
- Ethernet
- Multiprotocol
- X.25.

Link station statistics can be obtained with the DLC_QUERY_LS **ioctl** subroutine from various data link controls. These statistics include data link protocol counters. Each counter is reset by the DLC during the DLC_START_LS **ioctl** subroutine, and generally runs continuously until the link station is terminated and its storage is freed. If a counter reaches the maximum count, the count is frozen and no wrap around occurs.

The suggested counters to be provided by a DLC device manager are shown below. Some DLCs may wish to modify this set of counters based on the specific protocols being supported. For example, the number of rejects or receive-not-ready packets received might be meaningful.

Test Commands Sent

Contains a binary count of the **test** commands sent to the remote station by GDLC, in response to **test** commands issued by the user.

Test Command Failures

Contains a binary count of the **test** commands that did not complete properly due to problems such as the following:

- Invalid response
- Bad data compare
- Inactivity.

Test Commands Received

Contains a binary count of valid **test** commands received, regardless of whether the response is completed properly.

Sequenced Data Packets Transmitted

Contains a binary count of the total number of normal sequenced data packets that were transmitted to the remote link station.

Sequenced Data Packets Transmitted

Contains a binary count of the total number of normal sequenced data packets that were retransmitted to the remote link station.

Maximum Contiguous Retransmissions

Contains a binary count of the maximum number of times a single data packet has been retransmitted to the remote link station prior to acknowledgment. This counter is reset each time a valid acknowledgment is received.

Sequenced Data Packets Received

Contains a binary count of the total number of normal sequenced data packets that have been correctly received.

Invalid Packets Received

Contains a binary count of the number of invalid commands or responses received, including invalid control bytes, invalid I-fields, and overflowed I-fields.

Adapter Detected Receive Errors

Contains a binary count of the number of receive errors reported back from the device driver.

Adapter Detected Transmit Errors

Contains a binary count the number of transmit errors reported back from the device driver.

Receive Inactivity Time Outs

Contains a binary count of the number of receive time outs that have occurred.

Command Polls Sent

Contains a binary count of the number of command packets sent, that requested a response from the remote link station.

Command Repolls Sent

Contains a binary count of the total number of command packets that were retransmitted to the remote link station due to lack of response.

Command Contiguous Repolls

Contains a binary count of the number of times a single command packet was retransmitted to the remote link station due to lack of response. This counter is reset each time a valid response is received.

Understanding the DLC Error Log

Each DLC provides entries to the system error log whenever errors are encountered. To call the kernel error collector, use the **errsave** kernel service.

GDLC supports the GL product Network Management Alert Management architecture for reporting error conditions. The error conditions are reported using the GL product system error log using the error log daemon (**errdaemon**). Each error is defined with the following entries:

Error Type
Indicates the severity of the error. The three levels of severity are as follows:

- **Temporary** Indicates errors that do not force closure of the link station or SAP connections, but are logged for network analysis.
- **Permanent** Indicates errors that result in the closure of the individual link station, SAP, or the entire physical port due to their catastrophic nature.
- **Performance** Indicates errors such as queue overruns that are causing performance degradation due to retransmissions and other factors.

Error Description
Describes the failure.

Probable Cause
Describes what likely caused the failure.

Recommended Actions
Describes how to correct the problem. This is divided into operator (User), installation/set-up (Install), and resource (Failure) actions.

Detailed Data
Provides additional data obtained at the time of the error.

The user can obtain formatted error log data by issuing the **errpt** command. When used with the **-R** *DLCType* flag, the **errpt** command produces a detailed report of all the error log entries for the resource type indicated by the *DLCType* variable previously collected in the **/etc/rasconf** default file.

Valid values for the *DLCType* variable include:

dlcether Standard Ethernet datalink

dlc802.3 IEEE 802.3 Ethernet datalink

dlctoken Token-Ring datalink

dlcsdlc SDLC datalink

Alerts Some error conditions must generate specific error log formats so that Alert Vectors can be sent to the local network manager and possibly to a remote SNA host.

The format of each required Alert Vector can be found in Appendix A of IBM publication *SNA Format and Protocol Reference Manual: Management Services.*

Understanding the DLC Link Trace Facility

GDLC provides optional entries to a generic system trace channel as required by the GL product system Reliability/Availability/Serviceability. GDLC is defaulted with trace disabled in order to provide maximum performance and reduce the number of system resources utilized. For more information, see Understanding the DLC Local Area Network Monitor Trace.

Trace Channels

AIXv3 supports up to seven generic trace channels in operation simultaneously. A channel must be allocated by the user prior to activation of a link trace, whether it is being started in the DLC_START_LS **ioctl** operation or in the DLC_TRACE **ioctl** operation. This is accomplished with the **trcstart** and **trcon** subroutines.

Trace activity in the link station must be stopped by either halting the link station or by issuing an **ioctl**(DLC_TRACE, flags=0) operation to that station. See the DLC_TRACE ioctl operation for DLC. Once the link station has stopped tracing, the channel can be disabled using the **trcoff** subroutine and returned to the system using the **trcstop** subroutine.

Trace Reports

The user can obtain formatted trace log data by issuing the **trcrpt** command with the appropriate file name, such as:

```
trcrpt /tmp/link1.log
```

This example produces a detailed report of all the link trace entries in the **/tmp/link1.log** file, *if* a prior **trcstart** subroutine specified the **/tmp/link1.log** file as the (**-o**) name for the trace log.

Trace Entries

The subroutine call generated by GDLC to the kernel Generic Trace for each entry is:

#include <sys/trchkid.h>

void trcgenkt (*chan*, *hk_word*, *data_word*, *len*, *buf***)**
unsigned int *chan*, *hk_word*, *data_word*, *len*;
char **buf*;

where:

chan — Specifies the channel number for the trace session. This number is obtained from the **trcstart** subroutine.

hk_word — Contains the trace hook identifier as defined in the **/usr/include/sys/trchkid.h** header file.

Five types of link trace entries are registered using hook ID:

HKWD_SYSX_DLC_START	Start Link Station Completions
HKWD_SYSX_DLC_TIMER	Timeout Completions
HKWD_SYSX_DLC_XMIT	Transmit Completions
HKWD_SYSX_DLC_RECV	Receive Completions
HKWD_SYSX_DLC_HALT	Halt Link Station Completions.

data_word — Specifies trace data format field, varies depending on the hook ID. Each of these definitions can be found in the **/usr/include/sys/gdlextcb.h** header file.

The first half-word always contains the data link protocol field.

DLC_DL_SDLC	SDLC
DLC_DL_HDLC	HDLC
DLC_DL_BSC	BISYNC
DLC_DL_ASC	ASYNC
DLC_DL_PCNET	PC Network
DLC_DL_ETHER	Standard Ethernet
DLC_DL_802_3	IEEE 802.3
DLC_DL_TOKEN	Token-Ring
DLC_DL_QLLC	X.25 Qualified DLC.

The second half-word is as follows:

- On start or halt link station completions the second half-word contains the physical link protocol in use:

DLC_PL_EIA232	EIA-232D Telecommunications
DLC_PL_EIA366	EIA-366 Auto Dial
DLC_PL_X21	CCITT X.21 Data Network
DLC_PL_PCNET	PC Network Broadband
DLC_PL_ETHER	Standard Baseband Ethernet
DLC_PL_SMART	Smart Modem Auto Dial
DLC_PL_802_3	IEEE 802.3 Baseband Ethernet
DLC_PL_TBUS	IEEE 802.4 Token Bus
DLC_PL_TRING	IEEE 802.5 Token-Ring
DLC_PL_X25	X.25 Packet Network
DLC_PL_EIA422	EIA-422 Telecommunications
DLC_PL_V35	CCITT V.35 Telecommunications
DLC_PL_V25BIS	CCITT V.25 bis Autodial for Telecommunications.

- On time out completions the second half-word contains the type of time out occurrence:

DLC_TO_SLOW_POLL	Slow Station Poll
DLC_TO_IDLE_POLL	Idle Station Poll
DLC_TO_ABORT	Link Station Aborted
DLC_TO_INACT	Link Station Receive Inactivity
DLC_TO_FAILSAFE	Command Failsafe
DLC_TO_REPOLL_T1	Command Repoll
DLC_TO_ACK_T2	I-frame Acknowledgment.

- On transmit completions the second half-word is set to the data link control bytes being sent. Some transmit packets only have a single control byte, so the second control byte is not displayed in those cases.
- On receive completions the second half-word is set to the data link control bytes that were received. Some receive packets only have a single control byte, so the second control byte is not displayed in those cases.

len
Specifies the length in bytes of the entry specific data specified by the *buf* parameter.

buf
Specifies the pointer to the entry specific data that consists of one of the following completions:

Start Link Station	Link Station Diagnostic Tag, and the remote station's name and address.
Timeout	No specific data is recorded.
Transmit	Either the first 80 bytes or all the transmitted data, depending on the short/long trace option.

Receive	Either the first 80 bytes or all the received data, depending on the short/long trace option.
Halt Link Station	Link Station Diagnostic Tag, the remote station's name and address, and the result code.

Data Link Control (DLC) Reference Information

The following is a list of reference information for DLC:

- DLC Entry Points:
  - **dlcclose**
  - **dlcconfig**
  - **dlcioctl**
  - **dlcmpx**
  - **dlcopen**
  - **dlcread**
  - **dlcselect**
  - **dlcwrite**
- Kernel Services for DLC:
  - **fp_close**
  - **fp_ioctl**
  - **fp_open**
  - **fp_write**
- DLC Kernel Routines for DLC:
  - Datagram Data Received
  - Exception Condition
  - I-Frame Data Received
  - Network Data Received
  - XID Data Received.
- Subroutines Available for DLC:
  - DLC Extended Parameters for Subroutines:

    Extended Parameters for open
    Extended Parameters for read
    Extended Parameters for write
  - Application Subroutines for DLC:

    close
    ioctl
    open
    readx
    select
    writex
- DLC Operations:
  - ioctl Operations (op) for DLC.
  - Parameter Blocks by Operation for DLC.

Related Information

The **errpt** command.

The **errsave** kernel service, **trcgenkt** kernel service.

The **trcoff** subroutine, **trcon** subroutine, **trcstart** subroutine, **trcstop** subroutine.

The **icotl** subroutine for DLC, **open** subroutine for DLC.

Token–Ring Data Link Control

Token-Ring Data Link Control (DLCTOKEN) is a device manager that follows the generic interface definition (GDLC). This DLC device manager provides an access procedure for the transfer of four types of data over the IBM token ring: datagrams, sequenced data, identification data, and logical link controls. It also provides a pass-through capability that allows transparent data flow.

This access procedure relies on functions provided in the Token-Ring Device Handler Overview and the IBM Token-Ring Network 16/4 Co-Processor Busmaster Adapter to actually transfer data, with address checking, token generation, or frame check sequences.

The Token-Ring Data Link Control (DLCTOKEN) provides the following:

- DLCTOKEN device manager functions
- Name-discovery services
- Protocol support
- Direct network services
- Connection contention
- Link session initiation
- Link session termination
- Programming interfaces.

The DLCTOKEN device manager operates between two nodes on the Token-Ring LAN using IEEE 802.2 procedures and control information as defined in the *IBM Token-Ring LAN Formats and Protocols Manual (SC30-3374)*. This protocol support includes the following:

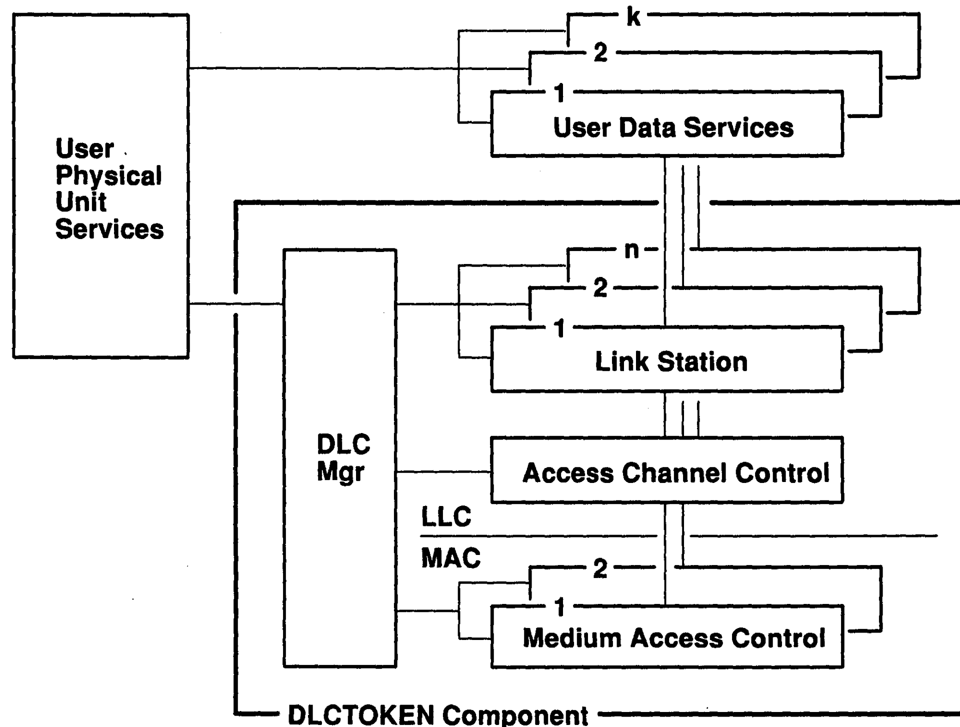
- Asynchronous Disconnected mode (ADM) and Asynchronous Balanced mode extended (ABME)
- Two-way simultaneous (full-duplex) data flow
- Multiple point-to-point logical attachments on the LAN using network and service access point addresses
- Peer-to-peer relationship with remote station
- Both name-discovery and address-resolve services
- Source-routing generation for up to eight bridge hops.

The Token-Ring Data Link Control (DLCTOKEN) provides full-duplex, peer-data transfer capabilities over an IBM Token-Ring local area network (LAN). The Token-Ring LAN must use the Token-Ring IEEE 802.5 medium access control (MAC) procedure and a superset of the IEEE 802.2 logical link control (LLC) protocol as described in the *IBM Token-Ring LAN Formats and Protocols Manual*.

Multiple Token-Ring adapters are supported, with a maximum of 254 service access point (SAP) users per adapter. A total of 255 link stations (LS) per adapter are supported, which are distributed among the active SAP users. Multiple ring segments can be accessed using IBM Token-Ring network bridge facilities, with up to eight consecutive ring segments supported between any two nodes.

The term logical link control (LLC) is used to describe the collection of manager, access channel, and link station subcomponents of a Generic Data Link Control (GDLC) component such as DLCTOKEN device manager. The following figure illustrates the DLCTOKEN Component Structure:

DLCTOKEN Component Structure



Each link station (LS) controls the transfer of data on a single logical link. The access channel performs multiplexing and demultiplexing for message units flowing from the LSs and manager to MAC. The DLC manager performs connection establishment and termination as well as LS creation and deletion. In addition, it routes commands to the proper station.

DLCTOKEN Device Manager Functions

The DLCTOKEN device manager and transport medium use two functional layers (MAC and LLC) to maintain reliable link-level connections, guarantee data integrity, negotiate exchanges of identification, and connectless connection-oriented services.

The Token-Ring adapter and the DLCTOKEN device handler are responsible for the following MAC functions:

- Ring-insertion protocol
- Token detection and creation
- Encoding and decoding the serial bit-stream data
- Checking received network, group, and functional addresses
- Routing of received frames based on the LLC/MAC indicator and using the destination SAP address if an LLC frame was received
- CRC checking and generation
- Frame delimiters, such as start/end delimiters and frame status field
- Failsafe time outs.

DLCTOKEN is responsible for additional medium access control functions, such as:

- Framing control fields on transmit frames
- Network addressing on transmit frames
- Routing information on transmit frames.

DLCTOKEN is also responsible for all the logical link control (LLC) functions:

- Remote connection services and bridge routing using the address-resolve and name-discovery procedures
- Sequencing of link stations on a given port
- Generating service access point (SAP) addresses on transmit frames
- Generating IEEE 802.2 LLC commands and responses on transmit frames
- Recognizing and routing of received frames to the proper service access point
- Servicing of IEEE 802.2 LLC commands and responses on receive frames
- Frame sequencing and retries
- Failsafe and inactivity time outs
- Reliability/Availability/Serviceability counters, error logs, and link trace.

Protocol Support

DLCTOKEN supports the logical link control (LLC) protocol and state tables described in the *IBM Token-Ring Local Area Network Format and Protocol Manual*, which contains the Local Area Network IEEE 802.2 Logical Link Control standard. Both address-resolve services and name-discovery services are supported for establishing remote connections. Also supported is a direct network interface that allows a user to transmit and receive unnumbered information packets directly through DLCTOKEN without any protocol handling done by the data link layer.

Station Type

A combined station is supported for a balanced (peer-to-peer) configuration on a logical point-to-point connection. This allows either station to asynchronously initiate the transmission of commands at any response opportunity. The sender in each combined station controls the receiver in the other station. Data transmissions then flow as primary commands, and acknowledgments and status flow as secondary responses.

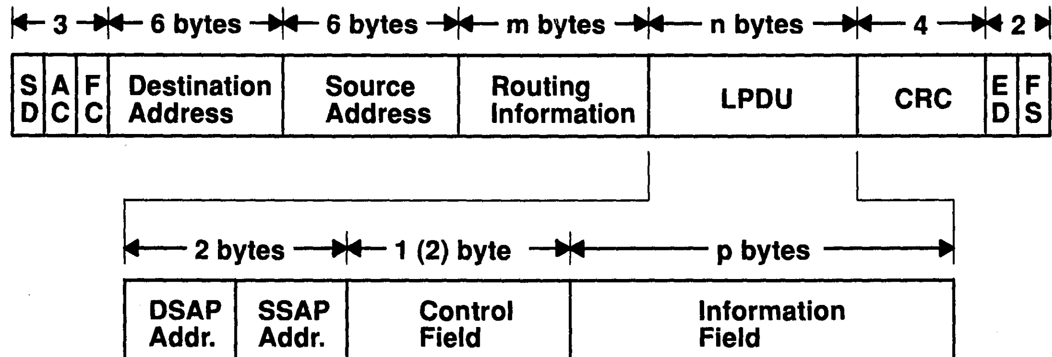
Response Modes

Both Asynchronous Disconnect mode (ADM) and Asynchronous Balanced mode extended (ABME) are supported. ADM is entered by default whenever a link session is initiated, and is switched to ABME only after the set Asynchronous Balanced mode extended (SABME) command sequence is complete. Once operating in ABME, information frames containing user data can be transferred. ABME then remains active until termination of the LLC session, which occurs due to the disconnect (DISC) command packet sequence or a major link error.

Token-Ring Data Packet

All communication between a local and remote station is accomplished by the transmission of a packet that contains the Token-Ring headers and trailers as well as an encapsulated LLC protocol data unit (LPDU). See the following figure that describes the Token-Ring data packet:

DLCTOKEN Frame Encapsulation



The Token-Ring data packet consists of the following:

SD	Starting delimiter
AC	Access control field
FC	Frame control field
LPDU	LLC protocol data unit
DSAP	Destination service access point (SAP) address field
SSAP	Source SAP address field
CRC	Cyclic redundancy check or frame check sequence
ED	Ending delimiter
FS	Frame status field
m bytes	Integer value greater than or equal to 0 and less than or equal to 18
n bytes	Integer value greater than or equal to 3 and less than or equal to 4064
p bytes	Integer value greater than or equal to 0 and less than or equal to 4060.

Note: SD, CRC, ED, and FS are added and deleted by the hardware adapter.

Name-Discovery Service

In addition to the standard IEEE 802.2 Common Logical Link Protocol support and address resolution services, DLCTOKEN also provides a name-discovery service that allows the operator to identify local and remote stations by name instead of by 6-byte physical addresses. Each port must have a unique name on the network of up to 20 characters. The character set used will vary depending on the user's protocol (SNA, for example, requires Character Set A). Additionally, each new SAP supported on a particular port may have a unique name if desired.

Each name is added to the network by broadcasting a find (local name) request when the new name is being introduced to a given network port. If no response other than an echo results from the find (local name) request after sending it the number of times specified, the physical link is declared opened, and the name is assigned to the local port and SAP. If another port on the network already has the name being added, a name found response is sent to the station that issued the find request, and the new attachment fails with a result code (`DLC_NAME_IN_USE`) indicating that a different name must be chosen.

Calls are established by broadcasting a find (remote name) request to the network and waiting for a response from the port with the specified name. Only those ports that have listen attachments pending, that receive colliding find requests, or that are already attached to the requesting remote station answer a find request.

Direct Network Services

Some users wish to handle their own unnumbered information packets on the network without the aid of the data link layer within DLCTOKEN. A direct network interface is provided that allows an entire packet to be generated and sent by a user once their SAP has been opened. This allows full control of every field in the data link header for each write issued. Also provided is the ability to view the entire packet contents on received frames. The only criteria for a direct network write are:

- The local SAP must be valid and opened.
- The data link control byte must indicate unnumbered information (0x03).

Connection Contention

Dual paths to the same nodes are detected by DLCTOKEN in one of two ways. If a call is in progress to a remote node that is also trying to call the local node, the incoming find (remote name) request is treated as if a local listen was outstanding. On the other hand, if a pending local listen has been acquired by a remote node's call, and the local user issues a call to that remote node after the LS is already active, a result code (`DLC_REMOTE_CONN`) is returned to the user along with the LS correlator of the attachment already active, so that the user can relink its attachment pointers.

Link Session Initiation

When a link station is opened DLCTOKEN is initialized at Open Link Station as a combined station in Asynchronous Disconnect mode (ADM). As a secondary/combined station, DLCTOKEN is in receive state waiting for a command frame from the primary/combined station. The command frames accepted by the secondary/combined station at this time are SABME (set Asynchronous Balanced mode extended), XID (exchange station identification), TEST (test link), UI (unnumbered information - datagram), and DISC (disconnect). Any other command frame is ignored. Once a SABME is received, the station is ready for normal data transfer, and I (information), RR (receive ready), RNR (receive not ready), and REJ (reject) frames are also accepted.

As a primary/combined station, DLCTOKEN can perform ADM XID exchanges and ADM TEST exchanges, send datagrams, or contact the remote into ABME. XID exchanges allow the primary/combined station to send out its station-specific identification to the secondary/combined station and obtain a response. Once an XID response is received, any attached information field is passed to the user for further action.

TEST exchanges allow the primary/combined station to send out a buffer of information that will be echoed by the secondary/combined station in order to test the integrity of the link.

Initiation of the normal data exchange mode (ABME) causes the primary/combined station to send a SABME to the secondary/combined station. Once sent successfully, the connection is said to be contacted and the user is notified. I-frames can now be sent and received between the linked stations.

Link Session Termination

DLCTOKEN can be terminated by the user or by the remote station.

- The user can cause normal termination by issuing a **DLC_HALT_LS** command to DLCTOKEN. This causes the primary/combined station to initiate a disconnect (DISC) packet sequence.
- Receive inactivity can be optioned to cause termination. This is useful in detecting a loss of connection in the middle of a session.
- The remote station can cause termination by sending a DISC command packet as a primary/combined station.
- Abnormal termination is caused by certain protocol violations or by resource outages.

DLCTOKEN Programming Interfaces

The Token-Ring Data Link Control (DLCTOKEN) conforms to the GDLC guidelines except where noted below. Additional structures and definitions for DLCTOKEN can be found in the **/usr/include/sys/trlextcb.h** header file.

Note: The **dlc** prefix is replaced with the **trl** prefix for DLCTOKEN.

trlclose DLCTOKEN is fully compatible with the **dlcclose** GDLC interface.

trlconfig DLCTOKEN is fully compatible with the **dlcconfig** GDLC interface. No initialization parameters are required.

trlmpx DLCTOKEN is fully compatible with the **dlcmpx** GDLC interface.

trlopen DLCTOKEN is fully compatible with the **dlcopen** GDLC interface.

trlread DLCTOKEN is compatible with the **dlcread** GDLC interface with the following conditions: The **readx** subroutines may have DLCTOKEN data link header information prefixed to the I-field being passed to the application. This is optional based on the **readx** subroutine *data link header length* extension parameter in the **gdl_io_ext** structure.

If this field is nonzero, DLCTOKEN copies the data link header and the I-field to user space, and sets the actual length of the data link header into the length field. If the field is zero, no data link header information is copied to user space. See the DLCTOKEN frame format for more details. It should be noted that there is always an 18-byte area for routing information regardless of whether routing was present.

Kernel **receive packet** function handlers always have the DLCTOKEN data link header information within the communications memory buffer (**mbuf**), and can locate it by subtracting the length passed (in the **gdl_io_ext** structure) from the **data offset** field of the **mbuf** structure.

trlselect DLCTOKEN is fully compatible with the **dlcselect** GDLC interface.

trlwrite DLCTOKEN is compatible with the **dlcwrite** GDLC interface, with the exception that network data may only be written as an unnumbered information (UI) packet and must have the complete data link header prefixed to the data. DLCTOKEN verifies that the local (source) SAP is enabled and that the control byte is UI (0x03). See the DLCTOKEN frame format for more details.

trlioctl DLCTOKEN is compatible with the **dlcioctl** GDLC interface, with conditions on the following operations:

DLC_ENABLE_SAP
DLC_START_LS
DLC_ALTER
DLC_ENTER_SHOLD
DLC_EXIT_SHOLD
DLC_ADD_GROUP
IOCINFO.

DLC_ENABLE_SAP

The **ioctl** argument structure for enabling a SAP (**dlc_esap_arg**) has the following specifics:

- The `grp_addr` (group address) field for token ring contains the four least significant bytes of the desired 6-byte group address. Only bits 1 through 31 are valid. Bit 0 (zero) is ignored. The most significant 2 bytes are automatically compared for 0xC000 by the adapter.
- The `func_addr_mask` (functional address mask) field must be the logical OR operation with the functional address on the adapter, which allows packets that are destined for specified functions to be received by the local adapter. Only bits 1 through 29 are valid. Bits 0, 30 and 31 are ignored. The most significant 2 bytes of the full 6-byte functional address are automatically compared for 0xC000 by the adapter.

  The following is an example of a NetBios functional address:

  ```
  To select the Netbios functional address of 0xC000_0000_0080,
  the functional address mask is set to 0x0000_0080.
  ```

  Note: No checks are made by DLCTOKEN as to whether a received packet was accepted by the adapter due to a pre-set network address, group address, or functional address.
- The `max_ls` (maximum link stations) field cannot exceed 255.
- The following are the common SAP flags that are not supported:

ENCD	SDLC serial encoding
NTWK	Teleprocessing network type
LINK	Teleprocessing link type
PHYC	Physical network call (teleprocessing)
ANSW	Teleprocessing autocall and autoanswer.

- Group SAPs are not supported, so the `num_grp_saps` (number of group SAPs) field must be set to 0 (zero).
- The `laddr_name` (local address/name) field and its associated length are only used for name discovery when the common SAP flag **ADDR** is set to 0. When resolve procedures are used (the **ADDR** flag set to one), DLCTOKEN obtains the local network address from the device handler, and not from the **dlc_esap_arg** structure.
- The `local_sap` (local service access point) field may be set to any value except the Null SAP (0x00) or the Discovery SAP (0xFC). Also, the low-order bit must be set to 0 (B'nnnnnnn0') to indicate an individual address.
- No protocol-specific data area is required for DLCTOKEN to enable a SAP.

DLC_START_LS

The **ioctl** argument structure for starting a link station (**dlc_sls_arg**) has the following specifics:

The following common link station flags that are *not* supported:

STAT Station type for SDLC
NEGO Negotiable station type for SDLC.

- The `raddr_name` (remote address/name) field is used only for outgoing calls when **DLC_SLS_LSVC** common link station flag is set active.
- The `maxif` (maximum I-field length) field may be set to any value greater than 0. See the DLCTOKEN frame format for more details. DLCTOKEN adjusts this value to a maximum of 4060 bytes if set too large.
- The `rcv_wind` (receive window) field may be set to any value from 1 (one) to 127. The recommended value is 127.
- The `xmit_wind` (transmit window) field may be set to any value from 1 to 127. The recommended value is 26.
- The `rsap` (remote SAP) field may be set to any value except the NULL SAP (0x00) or the name-discovery SAP (0xFC). Also, the low-order bit must be set to 0 (B'nnnnnnn0') to indicate an individual address.
- The `max_repoll` field may be set to any value from 1 to 255. The recommended value is 8.
- The `repoll_time` field is defined in increments of 0.5 seconds and may be set to any value from 1 to 255. The recommended value is 2, giving a time-out duration of 1 to 1.5 seconds.
- The `ack_time` (acknowledgment time) field is defined in increments of 0.5 seconds, and may be set to any value from 1 to 255. The recommended value is 1, giving a time-out duration of 0.5 to 1 second.
- The `inact_time` (inactivity time) field is defined in increments of 1 second and may be set to any value from 1 to 255. The recommended value is 48, giving a time-out duration of 48 to 48.5 seconds.
- The `force_time` (force halt time) field is defined in increments of 1 second and may be set to any value from 1 to 16383. The recommended value is 120, giving a time-out duration of approximately 2 minutes.
- A protocol-specific data area must be appended to the generic start link station argument (**dlc_sls_arg**). This structure provides DLCTOKEN with additional protocol-specific configuration parameters:

```
struct trl_start_psd
{
   uchar_t    pkt_prty;    /* ring access packet priority   */
   uchar_t    dyna_wnd;    /* dynamic window increment      */
   ushort_t   reserved;    /* currently not used            */
};
```

`pkt_prty` Specifies the ring access priority that the user wishes to reserve on transmit packets. Values of 0 to 3 are supported, where 0 is the lowest priority and 3 is the highest priority.

`dyna_wnd` Network congestion causes the local transmit window count to automatically drop to a value of 1. The dynamic window increment specifies the number of consecutive sequenced packets that must be acknowledged by the remote station before the local transmit window count can be incremented. This allows a gradual increase in network traffic after a period of congestion. This field may be set to any value from 1 to 255; the recommended value is 1.

DLC_ALTER

The **ioctl** subroutine argument structure for altering a link station (**dlc_alter_arg**) has the following specifics:

- The alter flags that are NOT supported:

  SM1, SM2 Set SDLC control mode.

- A protocol-specific data area must be appended to the generic alter link station argument structure (**dlc_alter_arg**). This structure provides DLCTOKEN with additional protocol-specific alter parameters.

```
#define TRL_ALTER_PRTY 0x80000000 /* alter packet priority      */
#define TRL_ALTER_DYNA 0x40000000 /* alter dynamic window incr. */
   struct trl_start_psd
{
   ulong_t   flags;      /* specific alter flags                */
   uchar_t   pkt_prty;   /* ring access packet priority value   */
   uchar_t   dyna_wnd;   /* dynamic window increment value      */
   ushort_t  reserved;   /* currently not used                  */
};
```

- The specific alter flags include:

  TRL_ALTER_PRTY Alter Priority. If set to 1, the `pkt_prty` value field replaces the current priority value being used by the LS. The LS must be started for this alter command to be valid.

  TRL_ALTER_DYNA Alter Dynamic Window. If set to 1, the `dyna_wnd` value field replaces the current dynamic window value being used by the LS. The LS must be started for this alter command to be valid.

- The `pkt_prty` value field specifies the new priority reservation value for transmit packets.

- The `dyna_wnd` value field specifies the new dynamic window value to control network congestion.

DLC_ENTER_SHOLD

The enter short hold option is not supported by DLCTOKEN.

DLC_EXIT_SHOLD

The exit short hold option is not supported by DLCTOKEN.

DLC_ADD_GROUP

The add group or multicast address option is not supported by DLCTOKEN.

IOCINFO

The **ioctype** variable returned is defined as a DD_DLC definition and the subtype returned is DS_DLCTOKEN.

Related Information

The **dlcclose** routine, **dlcconfig** routine, **dlcmpx** routine, **dlcopen** routine, **dlcread** routine, **dlcselect** routine, **dlcwrite** routine, **dlcioctl** routine.

The **ioctl** subroutine, **readx** subroutine.

IEEE 802.3 Ethernet Data Link Control (DLC8023) Overview on page 1–26, Standard Ethernet Data Link Control (DLCETHER) Overview on page 1–35,Synchronous Data Link Control (DLCSDLC) Overview on page 1–44, Qualified Logical Link Control (DLCQLLC) Overview on page 1–52.

IEEE 802.3 Ethernet Data Link Control

IEEE 802.3 Ethernet Data Link Control (DLC8023) is a device manager that follows the generic interface definition (GDLC). This DLC device manager provides a pass-through capability that allows transparent data flow and provides an access procedure for the transfer of four types of data over an Ethernet LAN: datagrams, sequenced data, identification data, and logical link controls.

This access procedure relies on functions provided in the Ethernet Kernel Device Driver and the IBM Ethernet Co-Processor Busmaster adapter to actually transfer data.

The IEEE 802.3 Ethernet Data Link Control (DLC8023) provides the following:

- DLC8023 device manager functions
- Protocol support
- Name-discovery services
- Direct network services
- Connection contention
- Link session initiation
- Link session termination
- Programming interfaces.

The DLC8023 device manager on an Ethernet LAN operates between two nodes: Media Access Control (MAC) procedures and IEEE 802.2 Logical Link Control (LLC) procedures. MAC and LLC procedures are defined in the IEEE Project 802 Local Area Network Standards. The specific state tables implemented can be found in the IBM Token-Ring LAN Formats and Protocols Manual. The DLC8023 device manager operating between these two nodes supports:

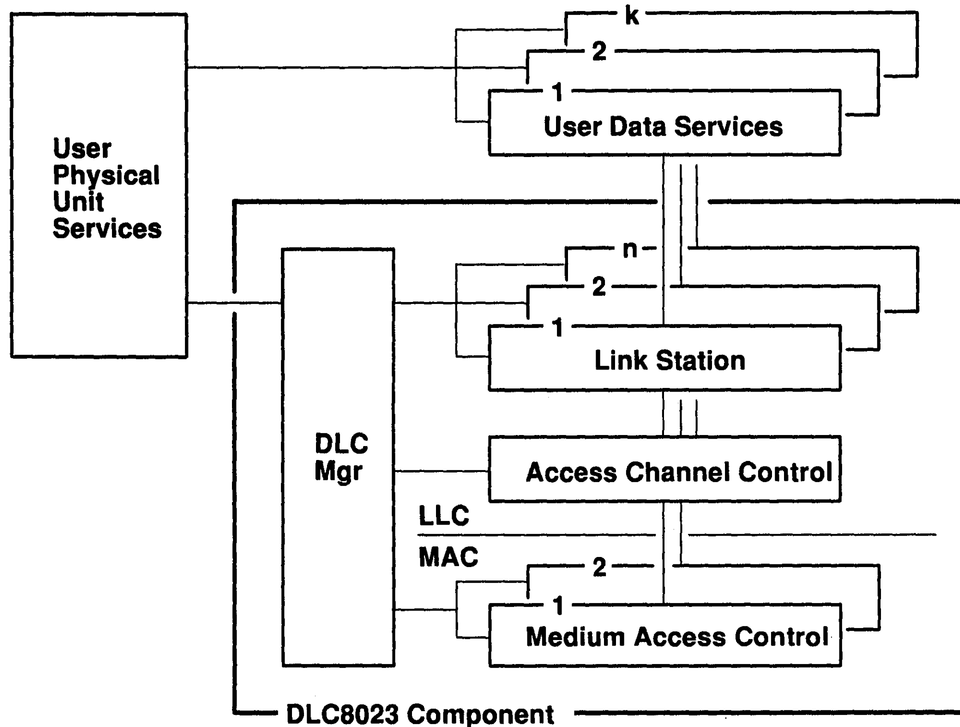
- Asynchronous Disconnected Mode (ADM) and Asynchronous Balanced Mode Extended (ABME)
- Two-way simultaneous (full-duplex) data flow
- Multiple point-to-point logical attachments on the LAN using network address and service access point address
- Peer-to-peer relationship with remote station
- Both name-discovery and address resolution services.

The DLC8023 device manager provides full-duplex, peer-data transfer capabilities over an Ethernet local area network. The Ethernet local area network must use the IEEE 802.3 CSMA/CD medium access control protocol and a superset of the IEEE 802.2 logical link control protocol.

Multiple adapters are supported, with a maximum of 255 logical attachments per adapter.

The term logical link control (LLC) is used to describe the collection of manager, access channel, and link station subcomponents of a Generic Data Link Control (GDLC) component such as the DLC8023 device manager. The following figure illustrates the DLC8023 Component Structure:

DLC8023 Component Structure



Each link station controls the transfer of data on a single logical link. The access channel performs multiplexing and demultiplexing for message units flowing from the link stations and manager to medium access control (MAC). The DLC manager performs connection establishment/termination, link station creation/deletion, and routing commands to the proper station.

DLC8023 Device Manager Functions

The DLC8023 device manager and transport medium use two functional layers, MAC and LLC to maintain reliable link-level connections, guarantee data integrity, negotiate exchanges of identification, and connectless connection-oriented services.

The Ethernet adapter and DLC8023 device handler are responsible for the following MAC functions:

- Managing the carrier sense multiple access with collision detection (CSMA/CD) algorithm
- Encoding and decoding the serial bit stream data
- Receiving network address checking
- Routing received frames based on the LPDU DSAP field
- Generating preamble
- Checking CRC and generation

- Failsafe time outs.

The DLC8023 device manager is also responsible for the following LLC functions.

- Remote connection services
- Sequencing of each link station on a given port
- Creating of the network addresses on transmit frames
- Creating of service access point (SAP) addresses on transmit frames
- Creating of IEEE 802.2 LLC commands and responses on transmit frames
- Recognizing and routing of received frames to the proper SAP
- Servicing of IEEE 802.2 LLC commands and responses on receive frames
- Frame sequencing and retries
- Failsafe and inactivity time outs
- Reliability/Availability/Serviceability counters, error logs, and link traces.

Protocol Support

DLC8023 supports the SNA logical link control (LLC) protocol and state tables described in the *IBM Token-Ring Local Area Network Format and Protocol Manual*, which contains the Local Area Network IEEE 802.2 Logical Link Control standard. Additional name discovery services have been added for establishing remote connections.

Station Type

A combined station is supported for a balanced (peer-to-peer) configuration on a logical point-to-point connection. This allows either station to asynchronously initiate the transmission of commands at any response opportunity. The data source in each combined station controls the data sink in the other station. Data transmissions then flow as primary commands, and acknowledgments and status flow as secondary responses.

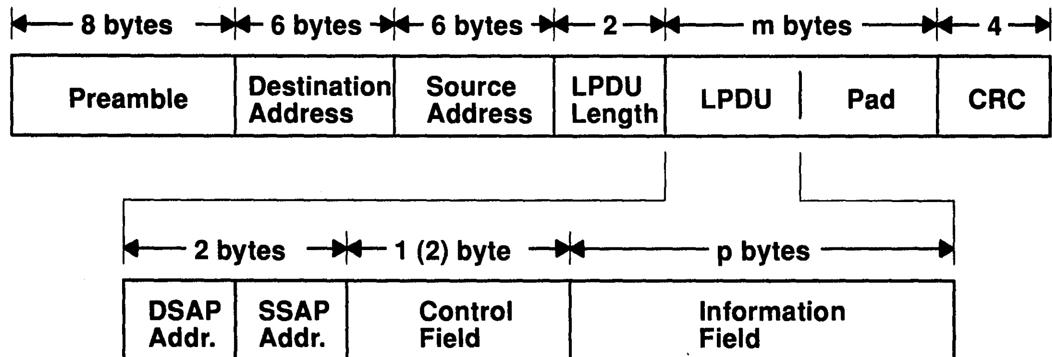
Response Modes

Both asynchronous disconnect mode (ADM) and asynchronous balanced mode extended (ABME) are supported. ADM is entered by default whenever a link session is initiated, and is switched to ABME only after the set asynchronous balanced mode extended (SABME) command sequence is complete. Once operating in the ABME command mode, information frames containing user data can be transferred. The ABME command mode then remains active until termination of the LLC session, which occurs due to the disconnect (DISC) command packet sequence or a major link error.

IEEE 802.3 Data Packet

All communication between a local and remote station is accomplished by the transmission of a packet that contains the IEEE 802.3 headers and trailers, and an encapsulated LLC protocol data unit (LPDU). See the following figure that describes the DLC8023 data packet:

DLC8023 Frame Encapsulation



The IEEE 802.3 data packet consists of the following fields:

LPDU	LLC protocol data unit
DSAP	Destination service access point address field
SSAP	Source service access point address field
CRC	Cyclic redundancy check or frame check sequence
m bytes	Integer value greater than or equal to 46 and less than or equal to 1500
p bytes	Integer value greater than or equal to 0 and less than or equal to 1496.

Note: Preamble and CRC are added and deleted by the hardware adapter.

Name Discovery Services

In addition to the standard IEEE 802.2 Common Logical Link Protocol support and the address resolution services, DLC8023 also provides a name service that allows the operator to identify local and remote stations by name instead of by 6-byte physical addresses. Each port must have a unique name of up to 20 characters on the network. The character set used will vary depending on the user's protocol (SNA, for example, requires Character Set A). Additionally, each new SAP supported on a particular port may have a unique name, if desired.

Each name is added to the network by broadcasting a find (local name) request when the new name is being introduced to a given network port. If no response results from the find (local name) request, after sending it the number of times specified, the physical link is declared opened, and the name is assigned to the local port and SAP. If another port on the network already has the name being added, a name found response is sent to the station that issued the find request, and the new attachment fails with a result code (`DLC_NAME_IN_USE`), indicating that a different name must be chosen.

Calls are established by broadcasting a find (remote name) request to the network and waiting for a response from the port with the specified name. Only those ports that have listen attachments pending, that receive colliding find requests, or that are already attached to the requesting remote station will answer a find request.

Direct Network Services

Some users wish to handle their own unnumbered information packets on the network without the aid of the data link layer within DLC8023. A direct network interface is provided that allows an entire packet to be generated and sent by a user once their service access point has been opened. This allows full control of every field in the data link header for each write issued. Also provided is the ability to view the entire packet contents on received frames. The only criteria for a direct network write are:

- The local SAP must be valid and opened.
- The data link control byte must indicate unnumbered information (0x03).

Connection Contention

Dual paths to the same nodes are detected by the DLC8023 device manager in one of two ways. If a call is in progress to a remote node that is also trying to call the local node, the incoming find (remote name) request is treated as if a local listen were outstanding.

On the other hand, if a pending local listen has been acquired by a remote node's call, and the local user issues a call to that remote node after the link session is already active, a result code (`DLC_REMOTE_CONN`) is returned to the user along with the link station correlator of the attachment already active, so that the user can relink its attachment pointers.

Link Session Initiation

The DLC8023 device manager is initialized at Open Link Station as a combined station in Asynchronous Disconnect Mode (ADM). As a secondary combined station, DLC8023 is in receive state waiting for a command frame from the primary combined station. The command frames accepted by the secondary combined station at this time are the SABME command (set asynchronous balanced mode extended), XID (exchange station identification), TEST (test link), UI (unnumbered information--datagram), and DISC (disconnect). Any other command frame will be ignored. Once a SABME command is received, the station is ready for normal data transfer, and I (Information), RR (receive ready), RNR (receive not ready), and REJ (reject) frames are also accepted.

As a primary combined station, the DLC8023 device manager can perform ADM XID exchanges, ADM TEST exchanges, send datagrams, or contact the remote into ABME. XID exchanges allow the primary combined station to send out its station specific identification to the secondary combined station and obtain a response. Once an XID response is received, any attached information field is passed to the user for further action.

TEST exchanges allow the primary combined station to send out a buffer of information that is echoed by the secondary combined station in order to test the integrity of the link.

Initiation of the normal data exchange mode (ABME) causes the primary combined station to send a SABME command to the secondary combined station. Once sent successfully, the connection is said to be *contacted*, and the user is notified. I-frames can now be sent and received between the linked stations.

Link Session Termination

The DLC8023 device manager can be terminated by the user or by the remote station in the following ways:

- The user can cause normal termination by issuing a **close link station** command to the DLC8023 device manager. This causes the primary/combined station to initiate a disconnect (DISC) packet sequence.

- The user can cause receive inactivity to cause termination. This is useful in detecting a loss of connection in the middle of a session.

- The remote station can cause termination by sending a disconnect (DISC) command packet as a primary combined station.
- Abnormal termination is caused by certain protocol violations or by resource outages.

DLC8023 Programming Interfaces

The IEEE 802.3 Ethernet Data Link Control (DLC8023) device manager conforms to the GDLC guidelines except where noted below.

Note: The **dlc** prefix is replaced with **e3l** prefix for the DLC8023 device manager.

e3lclose DLC8023 is fully compatible with the **dlcclose** GDLC interface.

e3lconfig DLC8023 is fully compatible with the **dlcconfig** GDLC interface. No initialization parameters are required.

e3lmpx DLC8023 is fully compatible with the **dlcmpx** GDLC interface.

e3lopen DLC8023 is fully compatible with the **dlcopen** GDLC interface.

e3lread DLC8023 is compatible with the **dlcread** GDLC interface, under the following condition:

The **readx** subroutines may have DLC8023 data link header information prefixed to the I-field being passed to the application. This is optional based on the **readx** subroutine *data link header length* extension parameter in the **gdl_io_ext** structure. If this field is nonzero, DLC8023 copies the data link header and the I-field to user space, and sets the actual length of the data link header into the length field. If the field is 0 (zero), no data link header information is copied to user space. See the DLC8023 frame format for more details.

Kernel **receive packet** function subroutine calls always have the DLC8023 data link header information within the communications memory buffer (**mbuf**), and can locate it by subtracting the length passed (in the **gdl_io_ext** structure) from the **mbuf** data offset field.

e3lselect DLC8023 is fully compatible with the **dlcselect** GDLC interface.

e3lwrite DLC8023 is compatible with the **dlcwrite** GDLC interface, with the exception that network data may only be written as an unnumbered information (UI) packet and must have the complete data link header prefixed to the data. DLC8023 verifies that the local (source) SAP is enabled and that the control byte is UI (0x03). See the DLC8023 frame format for more details.

e3lioctl DLC8023 is compatible with the **dlcioctl** GDLC interface, with conditions on the following operations:

DLC_ENABLE_SAP
DLC_START_LS
DLC_ALTER
DLC_ENTER_SHOLD
DLC_EXIT_SHOLD
DLC_ADD_GROUP
IOCINFO.

DLC_ENABLE_SAP

The **ioctl** argument structure for enabling a SAP (**dlc_esap_arg**) has the following specifics:

- The `grp_addr` (group address sometimes called multicast address) field must be set as specified in the IBM Ethernet Co-Processor Busmaster Adapter specifications. This is a 6-byte value that allows the local adapter to accept packets destined for a group of remote stations. An example of a group address is:

  `0x0900_2B00_0004.`

  Note: No checks are made by the DLC8023 device manager as to whether a received packet was accepted by the adapter due to burned in network address or group address.

- The `max_ls` (maximum link station) field cannot exceed 255.
- The following Common SAP Flags are *not* supported:

ENCD	SDLC serial encoding
NTWK	Teleprocessing network type
LINK	Teleprocessing link type
PHYC	Physical network call (teleprocessing)
ANSW	Teleprocessing auto call and answer.

- Group SAPs are *not* supported, so the `num_grp_saps` (number of group SAPs) field must be set to 0 (zero).
- The `laddr_name` (local address name) field and its associated length are only used for name-discovery when the common SAP flag **ADDR** is set to 0 (zero). When resolve procedures are used (the **ADDR** flag is set to one), the DLC8023 device manager obtains the local network address from the device handler, and not from the **dlc_esap_arg** structure.
- The `local_sap` (local SAP) field may be set to any value except the Null SAP (0x00) or the Discovery SAP (0xFC). Also, the low-order bit must be set to 0 (B'nnnnnnn0') to indicate an individual address.
- No protocol-specific data area is required for the DLC8023 device manager to enable a SAP.

DLC_START_LS

The **ioctl** argument structure for starting a link station (**dlc_sls_arg**) has the following specifies:

- The common link station flags are *not* supported:

STAT	Station type for SDLC
NEGO	Negotiable station type for SDLC.

- The `raddr_name` (remote address/name) field is used only for outgoing calls when **DLC_SLS_LSVC** common link station flag is set active.
- The `maxif` (maximum I-field length) field may be set to any value greater than 0. See the DLC8023 frame format for the specific byte lengths that are supported. DLC8023 adjusts this value to a maximum of 1496 bytes if set too large.
- The `rcv_wind` (receive window) field may be set to any value from 1 to 127. The recommended value is 127.

- The `xmit_wind` (transmit window) field may be set to any value from 1 to 127. The recommended value is 26.
- The `rsap` (remote SAP) field may be set to any value except the null SAP (0x00) or the discovery SAP (0xFC). Also, the low-order bit must be set to 0 (B'nnnnnnn0') to indicate an individual address.
- The `max_repoll` field may be set to any value from 1 to 255. The recommended value is 8.
- The `repoll_time` field is defined in increments of 0.5 seconds and may be set to any value from 1 to 255. The recommended value is 2, giving a time out duration of 1 to 1.5 seconds.
- The `ack_time` (acknowledgement time) field is defined in increments of 0.5 seconds, and may be set to any value from 1 to 255. The recommended value is 1, giving a timeout duration of 0.5 to 1 second.
- The `inact_time` (inactivity time) field is defined in increments of 1 second, and may be set to any value from 1 to 255. The recommended value is 48, giving a time out duration of 48 to 48.5 seconds.
- The `force_time` (force halt time) field is defined in increments of 1 second and may be set to any value from 1 to 16383. The recommended value is 120, giving a time out duration of approximately 2 minutes.
- There is no protocol-specific data area required for the DLC8023 device manager to start an LS (link station).

DLC_ALTER

The **ioctl** argument structure for altering a link station (**dlc_alter_arg**) has the following specifies:

- The following alter flags that are *not* supported:

  RTE Alter routing
  SM1, SM2 Set SDLC control mode.

- There is no protocol-specific data area required for DLC8023 to alter a link station.

DLC_ENTER_SHOLD

The enter short hold option is not supported by the DLC8023 device manager.

DLC_EXIT_SHOLD

The exit short hold option is not supported by the DLC8023 device manager.

DLC_ADD_GROUP

The add group or multi-cast address option is supported by the DLC8023 device manager as a 6-byte value as described above in the **DLC_ENABLE_SAP** (group address) ioctl operation.

IOCINFO

The **ioctype** variable returned is defined as a DD_DLC definition and the subtype returned is DS_DLC8023.

Related Information

The **dlcclose** routine, **dlcconfig** routine, **dlcmpx** routine, **dlcopen** routine, **dlcread** routine, **dlcselect** routine, **dlcwrite** routine, **dlcioctl** routine.

The **ioctl** subroutine, **readx** subroutine.

Token–Ring Data Link Control (DLCTOKEN) Overview on page 1–16, Standard Ethernet Data Link Control (DLCETHER) Overview on page 1–35, Synchronous Data Link Control (DLCSDLC) Overview on page 1–44, Qualified Logical Link Control (DLCQLLC) Overview on page 1–52.

Standard Ethernet Data Link Control

Standard Ethernet Data Link Control (DLCETHER) is a device manager that follows the generic interface definition (GDLC). This DLC device manager provides a pass-through capability that allows transparent data flow and provides an access procedure for the transfer of four types of data over a Standard Ethernet: datagrams, sequenced data, identification data, and logical link controls.

To actually transfer data, this access procedure relies on functions provided in the Ethernet Device Handler Overview and the IBM Ethernet Co-Processor Busmaster adapter. See also *RISC System/6000 Power Station and Power Server Hardware Technical Reference Options and Devices (SA23-2646).*

The Standard Ethernet Data Link Control (DLCETHER) provides the following:

- DLCETHER device manager functions
- Protocol support
- Name-discovery services
- Direct network services
- Connection contention
- Link session initiation
- Link session termination
- Programming interfaces.

The DLCETHER device manager on an Ethernet LAN operates between two nodes: media access control (MAC) procedures and IEEE 802.2 logical link control (LLC) procedures, as defined in the IEEE Project 802 Local Area Network Standards. The specific state tables implemented can be found in the IBM Token-Ring LAN Formats and Protocols Manual. This device manager operating between these two nodes supports:

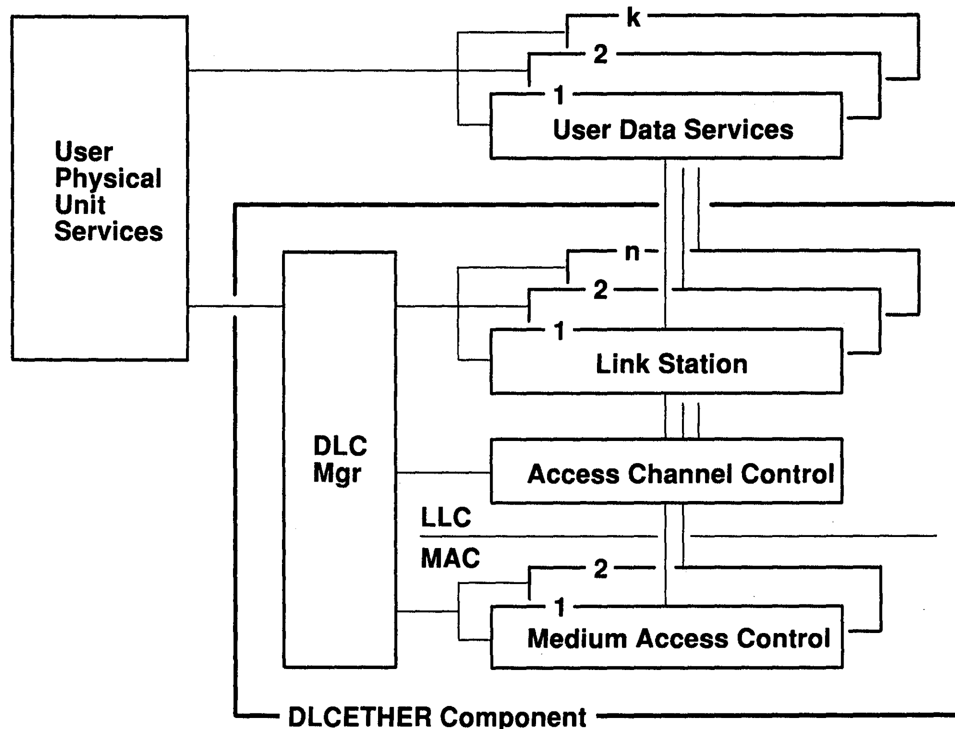
- Asynchronous Disconnected Mode (ADM) and Asynchronous Balanced Mode Extended (ABME)
- Two-way simultaneous (full-duplex) data flow
- Multiple point-to-point logical attachments on the LAN using network and service access point addresses
- Peer-to-peer relationship with remote station
- Both name-discovery and address resolution services.

The Ethernet Data Link Control provides full-duplex, peer data-transfer capabilities over a Standard Ethernet local area network, using the Xerox Ethernet CSMA/CD medium access control (MAC) protocol and a superset of the IEEE 802.2 logical link control (LLC).

Note: Multiple adapters are supported with a maximum of 255 logical attachments per adapter.

The term logical link control (LLC) is used to describe the collection of manager, access channel, and link station subcomponents of a Generic Data Link Control (GDLC) component such as DLCETHER device manager. The following figure illustrates the DLCETHER Component Structure:

DLCETHER Component Structure



Each link station controls the transfer of data on a single logical link. The access channel performs multiplexing and demultiplexing for message units flowing from the link stations and manager to medium access control (MAC). The DLC manager performs connection establishment and termination, link station creation and deletion, and routing commands to the proper station.

DLCETHER Device Manager Functions

The DLCETHER device manager and transport medium use two functional layers, medium access control (MAC) and logical link control (LLC) to maintain reliable link-level connections, guarantee data integrity, negotiate exchanges of identification, and connectless connection-oriented services.

The Ethernet adapter and the DLCETHER device handler are responsible for the following MAC functions:

- Managing the carrier sense multiple access with collision detection (CSMA/CD) algorithm
- Encoding and decoding the serial bit stream data
- Receiving network address checking
- Routing received frames based on the LLC type field
- CRC checking and generation
- Failsafe time outs.

The DLCETHER device manager is responsible for the following LLC functions, as well:

- Remote connection services
- Sequencing of each link station on a given port
- Creating of the network addresses on transmit frames
- Creating of service access point (SAP) addresses on transmit frames
- Creating of IEEE 802.2 LLC commands and responses on transmit frames
- Recognizing and routing of received frames to the proper SAP
- Servicing of IEEE 802.2 LLC commands and responses on receive frames
- Frame sequencing and retries
- Failsafe and inactivity time outs
- Reliability/Availability/Serviceability counters, error logs, and link traces.

Protocol Support

DLCETHER supports the SNA logical link control (LLC) protocol and state tables described in the *IBM Token-Ring Local Area Network Format and Protocol Manual*, which contains the Local Area Network IEEE 802.2 Logical Link Control standard. Additional direct-name services have been added for establishing remote connections.

Station Type

A combined station is supported for a balanced (peer-to-peer) configuration on a logical point-to-point connection. This allows either station to asynchronously initiate the transmission of commands at any response opportunity. The data source in each combined station controls the data sink in the other station. Data transmissions then flow as primary commands, and acknowledgments and status flow as secondary responses.

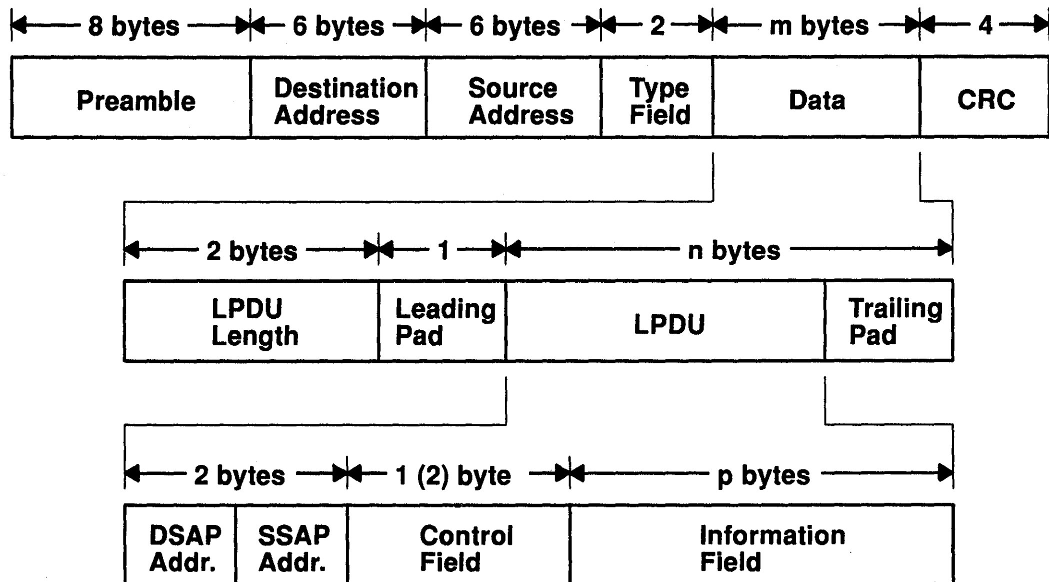
Response Modes

Both asynchronous disconnect mode (ADM) and asynchronous balanced mode extended (ABME) are supported. The ADM mode is entered by default whenever a link session is initiated, and is switched to ABME only after the set asynchronous balanced mode extended (SABME) command sequence is complete. Once operating in the ABME mode, information frames containing user data can be transferred. The ABME mode then remains active until termination of the LLC session, which occurs due to the disconnect (DISC) command sequence or a major link error.

ETHERNET Data Packet

All communication between a local and remote station is accomplished by the transmission of a packet that contains the token-ring headers and trailers, and an encapsulated LLC protocol data unit (LPDU). This packet format is specifically designed for IBM Systems Network Architecture (SNA) Logical Unit (LU) type 6.2 higher-level protocol, but other protocols can use this format as well. See the following figure that describes the Ethernet data packet:

DLCETHER Frame Encapsulation



The Ethernet data packet consists of the following fields:

LPDU LLC protocol data unit.
DSAP Destination service access point address field.
SSAP Source service access point address field.
CRC Cyclic redundancy check or frame check sequence.
m bytes Integer value greater than or equal to 46 and less than or equal to 1500.
n bytes Integer value greater than or equal to 43 and less than or equal to 1497.
p bytes Integer value greater than or equal to 0 (zero) and less than or equal to 1493.

Note: Preamble and CRC are added and deleted by the hardware adapter.

Name-Discovery Services

In addition to the standard IEEE 802.2 Common Logical Link Protocol support and the address resolution services, DLCETHER also provides a name service that allows the operator to identify local and remote stations by name instead of by 6-byte physical addresses. Each port must have a unique name on the network of up to twenty characters. The character set used varies depending on the user's protocol (SNA, for example, requires Character Set-A). Additionally, each new SAP supported on a particular port may have a unique name if desired.

Each name is added to the network by broadcasting a find (local name) request when the new name is being introduced to a given network port. If no response other than an echo results from the find (local name) request, after sending it the number of times specified, the physical link is declared opened, and the name is assigned to the local port and SAP. If

another port on the network already has the name being added, a name found response is sent to the station that issued the find request, and the new attachment fails with a result code (DLC_NAME_IN_USE), indicating that a different name must be chosen. Calls are established by broadcasting a find (remote name) request to the network and waiting for a response from the port with the specified name. Only those ports that have listen attachments pending, that receive colliding find requests, or that are already attached to the requesting remote station answers a find request.

Direct Network Services

Some users wish to handle their own unnumbered information packets on the network without the aid of the data link layer within DLCETHER. This results in protocol constraints from their individual service access points. A direct network interface is provided that allows an entire packet to be generated and sent by a user once the user's service access point has been opened. This allows full control of every field in the data link header for each write issued. Also provided is the ability to view the entire packet contents on received frames. The only criteria for a direct network write are that:

- The local SAP must be valid and opened.
- The data link control byte must indicate unnumbered information (0x03).

Connection Contention

Dual paths to the same nodes are detected by the DLCETHER device manager in one of two ways. If a call is in progress to a remote node that is also trying to call the local node, the incoming find (remote name) request is treated as if a local listen were outstanding. On the other hand, if a pending local listen has been acquired by a remote node's call and the local user issues a call to that remote node after the link session is already active, a result code (DLC_REMOTE_CONN) is returned to the user along with the link station correlator of the attachment already active. This allows the user to re-link attachment pointers.

Link Session Initiation

DLCETHER is initialized at Open Link Station as a combined station in Asynchronous Disconnect Mode (ADM). As a secondary/combined station, DLCETHER is in receive state waiting for a command frame from the primary combined station. The command frames accepted by the secondary combined station at this time are SABME (set asynchronous balanced mode extended), XID (exchange station identification), TEST (test link), UI (unnumbered information—datagram), and DISC (disconnect). Any other command frame will be ignored. Once a SABME command frame is received, the station is ready for normal data transfer, and the I (information), RR (receive ready), RNR (receive not ready), and REJ (reject) frames are also be accepted.

As a primary or combined station, DLCETHER can perform ADM XID exchanges, ADM TEST exchanges, send datagrams or contact the remote into the ABME command frame. XID exchanges allow the primary or combined station to send out its station specific identification to the secondary or combined station and obtain a response. Once an XID response is received, any attached information field is passed to the user for further action.

The TEST exchanges allow the primary or combined station to send out a buffer of information that is echoed by the secondary or combined station in order to test the integrity of the link.

Initiation of the normal data exchange mode (ABME) causes the primary combined station to send a SABME command frame to the secondary combined station. Once sent successfully, the connection is said to be *contacted*, and the user is notified. I-frames can now be sent and received between the partner stations.

Link Session Termination

The DLCETHER device manager can be terminated by the user or by the remote station in the following ways:

- The user can cause normal termination by issuing a DLC_HALT_LS command operation to the DLCETHER device manager. This will cause the primary/combined station to initiate a disconnect (DISC) command packet sequence.
- Receive inactivity can be optioned to cause termination. This is useful in detecting a loss of connection in the middle of a session.
- The remote station can cause termination by sending a DISC command packet as a primary combined station.
- Abnormal termination is caused by certain protocol violations or by resource outages.

DLCETHER Programming Interfaces

The Standard Ethernet Data Link Control (DLCETHER) conforms to the GDLC guidelines except where described in the following list:

Note: The **dlc** prefix is replaced with **edl** prefix for the DLCTOKEN device manager.

edlclose
DLCETHER is fully compatible with the GDLC interface, **dlcclose**.

edlconfig
DLCETHER is fully compatible with the GDLC interface, **dlcconfig**. No initialization parameters are required.

edlmpx
DLCETHER is fully compatible with the GDLC interface, **dlcmpx**.

edlopen
DLCETHER is fully compatible with the GDLC interface, **dlcopen**.

edlread
DLCETHER is compatible with the GDLC interface **dlcread** with the following conditions: The **readx** subroutines may have DLCETHER data link header information prefixed to the I-field being passed to the application. This is optional based on the **readx** subroutine *data link header length* extension parameter in the **gdl_io_ext** structure.

If this field is nonzero, DLCETHER copies the data link header and the I-field to user space, and sets the actual length of the data link header into the length field. If the field is zero, no data link header information is copied to user space. See the DLCETHER frame format for more details.

Kernel **receive packet** function subroutine calls always have the DLCETHER data link header information within the Communications memory buffer (**mbuf**), and can locate it by subtracting the length passed (in the **gdl_io_ext** structure) from the **mbuf** data offset field.

edlselect
DLCETHER is fully compatible with the GDLC interface, the **dlcselect**.

edlwrite
DLCETHER is compatible with the GDLC interface, **dlcwrite** with the exception that network data may only be written as an unnumbered information (UI) packet and must have the complete data link header prefixed to the data. DLCETHER verifies that the local (source) SAP is enabled and that the control byte is UI (0x03). See the DLCETHER frame format for more details.

edlioctl DLCETHER is compatible with the GDLC interface, **dlcioctl**, with conditions on the following operations:

DLC_ENABLE_SAP
DLC_START_LS
DLC_ALTER
DLC_ENTER_SHOLD
DLC_EXIT_SHOLD
DLC_ADD_GRP
IOCINFO

DLC_ENABLE_SAP

The **ioctl** argument structure for enabling a SAP (**dlc_esap_arg**) has the following specifics:

- The `grp_addr` field (group address--sometimes called multicast address) must be set as specified in the IBM Ethernet Co-Processor Busmaster adapter specifications. This is a 6-byte value that allows the local adapter to accept packets destined for a group of remote stations. An example of a group address follows:

  `0x0900_2B00_0004`

  Note: No checks are made by the DLCETHER device manager as to whether a received packet was accepted by the adapter due to a pre-set network address or group address.

- The `max_ls` (maximum link stations) field cannot exceed 255.
- The common SAP flags that are *not* supported:

  ENCD SDLC serial encoding
  NTWK Teleprocessing network type
  LINK Teleprocessing link type
  PHYC Physical network call (teleprocessing)
  ANSW Teleprocessing auto call and answer.

- Group SAPs are *not* supported, so the `num_grp_saps` (number of group SAPs) field must be set to 0.
- The `laddr_name` (local address and name) field and its associated length are only used for name-discovery when the common SAP flag field `ADDR` is set to 0 (zero). When resolve procedures are used (`ADDR` is set to one), DLCETHER obtains the local network address from the device handler, and not from the **dlc_esap_arg** structure.
- The `local_sap` field may be set to any value except the null SAP (0x00) or the discovery SAP (0xFC). Also, the low-order bit must be set to 0 (B'nnnnnnn0') to indicate an individual address.
- No protocol-specific data area is required for the DLCETHER device manager to enable a SAP.

DLC_START_LS

The **ioctl** argument structure for starting a link station (**dlc_sls_arg**) has the following specifics:

- The common link station flags that are *not* supported:

  STAT Station type for SDLC
  NEGO Negotiable station type for SDLC.

- The `raddr_name` (remote address/name) field is used only for outgoing calls when the **DLC_SLS_LSVC** common link station flag is set active.
- The `maxif` (max I-field) length may be set to any value greater than 0 (zero). See the frame format figure for specific byte lengths that are supported. The DLCETHER device manager adjusts this value to a maximum of 1493 bytes if set too large.
- The `rcv_wind` (receive window) field may be set to any value from 1 to 127. The recommended value is 127.
- The `xmit_wind` (transmit window) field may be set to any value from 1 to 127. The recommended value is 26.
- The `rsap` (remote SAP) field may be set to any value except the Null SAP (0x00) or the discovery SAP (0xFC). Also, the low-order bit must be set to 0 (B'nnnnnnn0') to indicate an individual address.
- The `max_repoll` field may be set to any value from 1 to 255. The recommended value is 8.
- The `repoll_time` field is defined in increments of 0.5 seconds and may be set to any value from 1 to 255. The recommended value is 2, giving a time out duration of 1 to 1.5 seconds.
- The `ack_time` (acknowledgement time) field is defined in increments of 0.5 seconds and may be set to any value from 1 to 255. The recommended value is 1, giving a time out duration of 0.5 to 1 second.
- The `inact_time` (inactivity time) field is defined in increments of 1 second, and may be set to any value from 1 to 255. The recommended value is 48, giving a time out duration of 48 to 48.5 seconds.
- The `force_time` (force halt time) field is defined in increments of 1 second, and may be set to any value from 1 to 16383. The recommended value is 120, giving a time out duration of approximately 2 minutes.
- There is no protocol-specific data area required for the DLCETHER device manager to start a link station.

DLC_ALTER

The **ioctl** argument structure for altering a link station (**dlc_alter_arg**) has the following specifics:

- The alter flags that are *not* supported:

  RTE Alter routing
  SM1, SM2 Set SDLC control mode.

- There is no protocol specific data area required for the DLCETHER device manager to alter a link station.

DLC_ENTER_SHOLD

The Enter Short Hold option is not supported by the DLCETHER device manager.

DLC_EXIT_SHOLD

The Exit Short Hold option is not supported by the DLCETHER device manager.

DLC_ADD_GRP

The add group or multicast address option is supported by the DLCETHER device manager as a 6-byte value as described above in **DLC_ENABLE_SAP** (group address) ioctl operation.

IOCINFO

The **ioctype** operation returned is DD_DLC definition and the subtype returned is **DS_DLCETHER**.

Related Information

The **dlcclose** routine, **dlcconfig** routine, **dlcmpx** routine, **dlcopen** routine, **dlcread** routine, **dlcselect** routine, **dlcwrite** routine, **dlcioctl** routine.

The **ioctl** subroutine, **readx** subroutine.

The Token-Ring Data Link Control (DLCTOKEN) Overview on page 1–16, The IEEE 802.3 Ethernet Data Link Control (DLC8023) Overview on page 1–26, The Synchronous Data Link Control (DLCSDLC) Overview on page 1–44, The Qualified Logical Link Control (DLCQLLC) on page 1–52.

The *RISC System/6000 Power Station and Power Server Hardware Technical Reference Options and Devices (SA23-2646).*

Synchronous Data Link Control

Synchronous Data Link Control (DLCSDLC) is one of the generic data link controls. It provides the access procedure for transparent and code-independent information interchange across teleprocessing and data networks, as defined in the IBM SDLC Concepts, GA27-3039 document. The subset of the SDLC architecture supported by DLCSDLC is as follows:

- Normal disconnected mode (NDM) and Normal Response mode (NRM)
- Two-way alternate (half-duplex) data flow
- Secondary station point-to-point, multipoint, and multi-multipoint configurations
- Primary station point-to-point and multipoint configurations
- Modulo 8 transmit and receive sequence counts
- Nonextended (single-byte) station address.

DLCSDLC Device Manager Functions

SDLC is split between a physical adapter with its associated device handler and a data link control (DLC) component. The DLCSDLC device manager is responsible for the following SDLC functions:

- Information frame sequencing
- Creation of address and control for transmit frames
- Service of control for receive frames
- Repoll and inactivity time outs
- Frame-reject generation
- Transmit windows
- Reliability/Availability/Serviceability counters, error logs, and link traces.

The device handler and adapter are jointly responsible for the remaining SDLC functions:

- Station address recognition
- NRZI/NRZ encoding and decoding
- Zero bit insertion and deletion
- Frame-check sequence generation and checking
- Flag and pad generation and deletion
- Interframe time fill
- Transmit failsafe timer
- Line-attachment protocols, such as RS232C, X.21, and Smartmodem
- Failsafe time out
- Autoresponse for nonproductive supervisory command frames.

DLCSDLC Protocol Support

Station Types

Two station types are supported:

- The primary station responsible for control of data interchange on the link
- The secondary, or subordinate, station on the link.

Transmission Frames

All communication between the local and remote stations is accomplished by the transmission of frames. The SDLC frame format consists of the following fields:

- Unique Flag Sequence (B'01111110') 1 byte
- Station Link Address Field 1 byte
- Control Field 1 byte
- Information Field n bytes
- Frame Check Sequence 2 bytes
- Unique Flag Sequence (B'01111110') 1 byte.

There are three kinds of SDLC frames: information, supervisory, and unnumbered. Information frames transport sequenced user data between the local and remote stations. Supervisory frames carry control parameters relative to the sequenced data transfer. Unnumbered frames transport the controls relative to nonsequenced transfers.

Response Modes

Both Normal Disconnect Mode (NDM) and Normal Response Mode (NRM) are supported. NDM is entered by default whenever a session is initiated, and is switched to NRM only after completion of the Set Normal Response Mode/Unnumbered Acknowledge (SNRM/UA) command sequence. Once operating in NRM, information frames containing user data can be transferred. NRM then remains active until termination of the SDLC session, which occurs due to the Disconnect/Unnumbered Acknowledge (DISC/UA) command sequence or a major link error. Once termination is complete, SDLC activity halts, and the NDM/NRM modes are not re-entered until another session is initiated.

Station Link Address Field

The station link address field supported is nonextended and consists of either the all-stations (Broadcast) address or a single unique 8-bit value other than the all-zeros (Null) address. The secondary station's address may be any value between 1 and 254, inclusive. Address value 255 (Broadcast) is used only by the primary station for initial contact of a point-to-point secondary station type, where the secondary's address is unknown. Once contact has been made, the secondary station's returned address is used exclusively for the remainder of the session.

Control Field (Commands Supported)

All commands are generated by the primary station for the secondary station. Each command carries the poll indicator to request immediate response, except when sending multiple information frames. Information frames that are concatenated have the poll indicator turned on in the last frame of the burst. The commands supported are as follows:

Information Sends sequenced user data from the primary station to the secondary station, as well as acknowledging any received information frames.

Receive Ready

Indicates that receive storage is available, and acknowledges any received information frames.

Receive Not Ready

Indicates receive storage is not available and acknowledges any received information frames.

Disconnect Requests the logical and physical disconnection of the link. An unnumbered command.

Set Normal Response Mode
Requests entry into Normal Response Mode, and resets the information sequence counts.This command is unnumbered.

Test Solicits an echoed TEST response from the secondary station and may carry an optional information field. This command is unnumbered.

Exchange Station Identification
Solicits an XID response that contains either the station identification of the secondary station or link negotiation information allowing the alteration of the primary/secondary relationship by the user. This command is unnumbered.

Control Field (Responses Supported)

All responses are generated by the secondary station for the primary station. Each response carries the final indicator to specify send completion, except when sending multiple information frames. Information frames that are concatenated have the final indicator on in the last frame of the burst. The responses supported are as follows:

Information Sends sequenced user data from the secondary station to the primary station. It also acknowledges any received information frames.

Receive Ready Indicates receive storage is available and acknowledges any received information frames.

Receive Not Ready
Indicates receive storage is not available and acknowledges any received information frames.

Frame Reject Indicates that the secondary station detects a problem in a command frame that otherwise had a valid frame check sequence in Normal Response Mode. An unnumbered response. The types of frame reject supported are:

Type 01H Invalid or nonimplemented command received.

Type 03H Invalid information field attached to command received.

Type 04H I-field exceeded buffer capacity (this value is not supported by DLCSDLC). Each overflowed receive buffer is passed to the user with an indication of overflow.

Type 08H The received Number Received (NR) sequence count is out of range.

Disconnected Mode
Indicates that the secondary station is in Normal Disconnect Mode. This response is unnumbered.

Unnumbered Acknowledge
Acknowledges receipt of the Set Normal Response Mode or Disconnect commands that were sent by the primary station. This response is unnumbered.

Test Echos the TEST command frame sent by the primary station, and carries the information field received only if sufficient storage is available. This response is unnumbered.

Exchange Station Identification
Contains the station identification of the secondary station. This response is unnumbered.

DLCSDLC Programming Interfaces

The SDLC Data Link Control (DLCSDLC) conforms to the GDLC guidelines except where noted in the following list. Additional structures and definitions for DLCSDLC can be found in the **/usr/include/sys/sdlextcb.h** header file.

Note: The **dlc** prefix is replaced with the **sdlc** prefix for DLCSDLC.

sdlclose DLCSDLC is fully compatible with the **dlcclose** GDLC interface.

sdlconfig DLCSDLC is fully compatible with the **dlcconfig** GDLC interface. No initialization parameters are required.

sdlmpx DLCSDLC is fully compatible with the **dlcmpx** GDLC interface.

sdlopen DLCSDLC is fully compatible with the **dlcopen** GDLC interface, with the following condition: only one open is allowed per port. This open can come from either an application or kernel user, but multiple users cannot share the same port.

sdlread DLCSDLC is compatible with the **dlcread** GDLC interface, with the following condition: network data is defined as any data received from the data communications equipment (DCE) that is not specific to the SDLC session protocol. Examples are X.21 call-progress signals or Smartmodem call-establishment messages. This data must be interpreted differently, depending on the physical attachment in use.

Datagram receive data is not supported.

sdlselect DLCSDLC is fully compatible with the **dlcselect** GDLC interface.

sdlwrite DLCSDLC is compatible with the **dlcwrite** GDLC interface, with the exception that network data and datagram data are not supported in the send direction. Network data such as X.21 or Smartmodem call establishment data is sent using the DLC_ENABLE_SAP **ioctl** operation.

sdlioctl DLCSDLC is compatible with the **dlcioctl** GDLC interface, with conditions on the following operations:

DLC_ADD_GRP
DLC_ALTER
DLC_ENABLE_SAP
DLC_ENTER_SHOLD
DLC_EXIT_SHOLD
DLC_START_LS
IOCINFO

DLC_ADD_GRP

The add group or multicast address option is not supported by DLCSDLC.

DLC_ALTER

The **ioctl** subroutine argument structure for altering a link station (**dlc_alter_arg**) has the following specifics:

- The alter flags that are *not* supported:

  AKT Alter acknowledgment time out
  RTE Alter routing.

- The `act_time` (acknowledge time out) field is ignored.
- The routing data field is ignored.
- No protocol-specific data area is required for DLCSDLC to alter its configuration.

DLC_ENABLE_SAP

DLCSDLC can support only a single DLC_ENABLE_SAP **ioctl** operation per port. All additional DLC_ENABLE_SAP **ioctl** operations are rejected.

The **ioctl** subroutine argument structure for enabling a SAP (**dlc_esap_arg**) has the following specifics:

- The `func_addr_mask` (function address mask) field is not supported.
- The `grp_addr` (group address) field is *not* supported.
- The `max_ls` (maximum link stations) field cannot exceed 254 on a multidrop primary link and cannot exceed 1 on a point-to-point or multidrop secondary link.
- The following common SAP flags are *not* supported:

  ADDR Local address or name indicator.

- The `laddr_name` (local address/name) field is *not* supported, so the length of local address/name field and local address/name field are ignored.
- Group SAPs are *not* supported, so the `num_grp_saps` (number of group SAPs) and `grp_sap` (group SAP–n) fields are ignored.
- The `local_sap` field is *not* supported and is ignored.
- The protocol specific data area is identical to the start device structure required by the Multiprotocol device handler. See the **/usr/include/sys/mpqp.h** header file and the **t_start_dev** structure for more details.

DLC_ENTER_SHOLD

The enter short hold option is not currently supported by DLCSDLC.

DLC_EXIT_SHOLD

The exit short hold option is not currently supported by DLCSDLC.

DLC_START_LS

DLCSDLC supports up to 254 concurrent link stations (LS) on a single port when it operates as a multidrop primary node. Only one LS can be started when DLCSDLC operates as a secondary node or when operating on a point-to-point connection.

The **ioctl** subroutine argument structure for starting an LS (**dlc_sls_arg**) has the following specifics:

- The following common flags are *not* supported:

  LSVC Link station virtual call is ignored.

  ADDR Address indicator must be set to 1 to indicate that no name-discovery services are provided.

- The `len_raddr_name` (length of remote's address/name) field must be set to 1.
- The `raddr_name` (remote's address/name) field is the 1–byte station address of the remote node in hex.
- The `maxif` (maximum I–field length) field may be set to any value greater than 0 (zero). DLCSDLC adjusts this value to a maximum of 4094 bytes if set too large.
- The `rcv_wind` (maximum receive window) field may be set to any value from 1 to 7. The recommended value is 7.
- The `xmit_wind` (maximum transmit window) field may be set to any value from 1 to 7. The recommended value is 7.
- The `rsap` (remote SAP) field is ignored.
- The `rsap_low` (remote SAP low range) field is ignored.
- The `rsap_high` (remote SAP high range) field is ignored.
- The `max_repoll` field may be set to any value from 1 to 255. The recommended value is 15.
- The `repoll_time` field is defined in increments of 0.1 second and may be set to any value from 1 to 255. The recommended value is 30, giving a time-out duration of approximately 3 seconds.
- The `ack_time` (acknowledgment time) field is ignored.
- The `inact_time` (inactivity time) field is defined in increments of 1 second and may be set to any value from 1 to 255. The recommended value is 30, giving a time-out duration of approximately 30 seconds.
- The `force_time` (force halt time) field is defined in increments of 1 second and may be set to any value from 1 to 16383. The recommended value is 120, giving a time-out duration of approximately 2 minutes.

- The following protocol-specific data area must be appended to the generic start link station argument structure (**dlc_sls_arg**). This structure provides DLCSDLC with additional protocol-specific configuration parameters:

```
struct    sdl_start_psd
{
uchar_t   duplex;                                                /*
link station xmit/receive capability                              */
uchar_t   secladd;          /* secondary station local address  */
uchar_t   prirpth;          /* primary repoll timeout threshold */
uchar_t   priilto;          /* primary idle list timeout        */
uchar_t   prislto;          /* primary slow list timeout        */
uchar_t   retxct;           /* retransmit count ceiling         */
uchar_t   retxth;           /* retransmit count threshold       */
uchar_t   reserved;         /* currently not used               */
 };
```

The protocol-specific parameters are:

duplex
Link station transmit-receive capability. This field must be set to 0, indicating two–way alternating capability.

secladd
This field specifies the secondary station link address of the local station. If the local station is negotiable, this address is used only if the local station becomes a secondary from role negotiation.

prirpth
Primary repoll threshold. This field specifies the number of contiguous repolls that will cause the local primary to log a temporary error. Any value from 1 to 100% may be specified. The recommended value is 10%.

priilto
Primary idle list time out. If the primary station has specified the Hold Link on Inactivity parameter and then discovers that a secondary station is not responding, the primary station places that secondary on an *idle list*. The primary station polls a station on the idle list less frequently than the other secondary stations to avoid tying up the network with useless polls. This field sets the amount of time (in seconds) that the primary station should wait between polls to stations on the idle list. Any value from 1 to 255 may be specified. The recommended value is 60, giving a time-out duration of approximately 60 seconds.

prislto
Primary slow list time out. When the primary station discovers that communication with a secondary station is not productive, it places that station on a *slow list*. The primary station polls a station on the slow list less frequently than the other secondary stations to avoid tying up the network with useless polls. This field sets the amount of time (in seconds) that the primary station should wait between polls to stations on the slow list. Any value from 1 to 255 may be specified. The recommended value is 20, giving a time-out duration of approximately 20 seconds.

retxct
Retransmit count. This field specifies the number of contiguous information frame bursts containing the same data that the local station retransmits before it declares a permanent transmission error. Any value from 1 to 255 may be specified. The recommended value is 10.

retxth Retransmit threshold. This field specifies the number of information frame retransmissions allowed as a percentage of total information frame tranmission (sampled only after a block of information frames has been sent). The specified percentage is the maximum rate of retransmissions allowed above which the system declares that a temporary error has occurred. Any value from 1 to 100% may be specified. The recommended value is 10%.

IOCINFO

The **ioctype** variable is defined as a DD_DLC definition and the subtype returned is DS_DLCSDLC.

Asynchronous Function Subroutine Calls

Datagram data received is not supported, and the **rcvd_fa** function is never called by DLCSDLC.DLCSDLC is compatible with each of the other asynchronous function subroutine calls for the kernel user.

Related Information

The **dlcclose** routine, **dlcconfig** routine, **dlcmpx** routine, **dlcopen** routine, **dlcread** routine, **dlcselect** routine, **dlcwrite** routine, **dlcioctl** routine.

The **ioctl** subroutine.

Token–Ring Data Link Control (DLCTOKEN) Overview on page 1–16, IEEE 802.3 Ethernet Data Link Control (DLC8023) Overview on page 1–35, Standard Ethernet Data Link Control (DLCETHER) Overview on page 1–35, Qualified Logical Link Control (DLCQLLC) Overview on page 1–52.

Qualified Logical Link Control

Qualified logical link control (DLCQLLC) is one of the generic data link controls. It provides the access procedure for attachment to X.25 packet switching networks.

DLCQLLC for AIX provides full support for the 1980 and 1984 versions of the CCITT recommendation relevant to SNA-to-SNA connections. It allows point-to-point connections to be established over an X.25 network between a pair of primary and secondary link stations. It supports modulo 8/128 packet sequence numbering and the following X.25 optional facilities:

- Closed user groups
- Recognized private operating agencies
- Network user identification
- Reverse charging
- Packet size negotiation
- Window size negotiation
- Throughput class negotiation.

DLCQLLC provides two-way alternate (half-duplex) data flow over switched or permanent virtual circuits.

DLCQLLC Device Manager Functions

DLCQLLC, as described in the X.25 Interface for Attaching SNA Nodes to Packet-Switch Data Networks (GA27-3345) and X.25 1984 Interface Architectural Reference (SC30-3409), is split between a physical adapter with its associated device handler and a DLC component. The data link control component is responsible for the following QLLC functions:

- Creation of address and control for transmit frames
- Service of control for receive frames
- Repoll and inactivity time outs
- Frame-reject generation
- Facility negotiation.

The data link control and device handler components are jointly responsible for the following:

- Establishment of an X.25 virtual circuit
- Clearing of an X.25 virtual circuit
- Notification of exceptional conditions to higher levels
- Reliability/Availability/Serviceability counters, error logs, and link traces.

The device handler and adapter are jointly responsible for the following:

- Packetization of I-frames
- Packet sequencing
- LAPB procedures as defined by CCITT recommendation X.25
- Physical line attachment protocols.

DLCQLLC Programming Interfaces

The QLLC Data Link Control (DLCQLLC) conforms to the GDLC guidelines except where noted below.

Note: The **dlc** prefix is replaced with **qlc** prefix for DLCQLLC.

qlcclose DLCQLLC is fully compatible with the **dlcclose** GDLC interface.

qlcconfig DLCQLLC is fully compatible with the **dlcconfig** GDLC interface. No initialization parameters are required.

qlcmpx DLCQLLC is fully compatible with the **dlcmpx** GDLC interface.

qlcopen DLCQLLC is fully compatible with the **dlcopen** GDLC interface.

qlcread DLCQLLC is compatible with the **dlcread** GDLC interface except that network data and datagram receive data are not supported.

qlcselect DLCQLLC is fully compatible with the **dlcselect** GDLC interface.

qlcwrite DLCQLLC is compatible with the **dlcwrite** GDLC interface with the exception that network data and datagram data are not supported.

qlcioctl DLCQLLC is compatible with the **dlcioctl** GDLC interface with the following conditions on the following operations:

DLC_ADD_GRP
DLC_ALTER
DLC_ENABLE_SAP
DLC_ENTER_SHOLD
DLC_EXIT_SHOLD
IOCINFO
DLC_START_LS

DLC_ADD_GRP

The add group or multicast address option is not supported by DLCQLLC.

DLC_ALTER

The **ioctl** subroutine argument structure for altering a link station (**dlc_alter_arg**) has the following differences:

- The alter flags that are *not* supported are:

  AKT Alter acknowledgment time out
  RTE Alter routing
  XWIN Alter transmit window size.

- The acknowledge time out field is ignored.
- The routing data field field is ignored.
- The transmit window size field is ignored.
- No protocol-specific data area is required for DLCQLLC to alter its configuration.

DLC_ENABLE_SAP

The **ioctl** subroutine argument structure for enabling a SAP (**dlc_esap_arg**) has the following differences:

- The function address mask field is *not* supported.
- The group address field is *not* supported.
- The common SAP flags are *not* supported:
- Group SAPs are *not* supported, so the number of group SAPs and group SAP–n fields are ignored.
- The local SAP field is *not* supported and is ignored.
- The protocol-specific data area is not required.

DLC_ENTER_SHOLD

The enter short hold option is not supported by DLCQLLC.

DLC_EXIT_SHOLD

The exit short hold option is not supported by DLCQLLC.

IOCINFO

The **ioctype** variable is defined as a DD_DLC definition and the subtype is DS_DLCQLLC.

DLC_START_LS

DLCQLLC supports up to 255 concurrent link stations (LS) on a single SAP.

The **ioctl** subroutine argument structure for starting an LS (**dlc_sls_arg**) has the following differences:

- The following common link station flag is *not* supported:

  ADDR The address indicator flag is ignored.

- The remote's address/name field is the network user address of the remote node.
- If the CCITT attribute is set to 1980 when configuring the X.25 adapter, the `rcv_window` (max receive window) field may be set to any value from 1 to 7. If the CCITT configuration attribute is set to 1984, the `rcv_window` (max receive window) field may be set to any value from 1 to 128.
- If the CCITT attribute is set to 1980 when configuring the X.25 adapter, the `xmit_wind` (max transmit window) field may be set to any value from 1 to 7. If the CCITT configuration attribute is set to 1984, the `xmit_wind` (max transmit window) field may be set to any value from 1 to 128.
- The RSAP (remote SAP) field is ignored.
- The RSAP low (remote SAP low range) field is ignored.
- The RSAP high (remote SAP high range) field is ignored.
- The repoll time field is defined in increments of 1 second.
- The ack time (acknowledgment time) field is ignored.
- A protocol-specific data area must be appended to the generic start link station argument (**dlc_sls_arg**).

Example of Protocol-Specific Configuration Parameters

The following is an example of a structure that provides DLCQLLC with additional protocol-specific configuration parameters:

```
struct qlc_start_psd
{
  char     listen_name[8];
  unsigned short support_level;
  struct   sna_facilities_type facilities;
};
```

The protocol-specific parameters are:

`listen_name` The name of the entry in the X.25 routing list that specifies the characteristics of incoming calls. This field is used only when a station is listening, that is, when the **LSVC** flag in the **dlc_sls_arg** argument structure is 0.

`support_level` The version of CCITT recommendation X.25 to be supported. It must be the same as or earlier than the CCITT attribute specified for the X.25 adapter, using SMIT.

`facilities` A structure that contains the X.25 facilities required for use on the virtual circuit for the duration of this attachment.

The Facilities Structure

The following is the SDLC facilities structure:

```
struct sna_facilities_type
{
  unsigned       facs:1;
  unsigned       rpoa:1;
  unsigned       psiz:1;
  unsigned       wsiz:1;
  unsigned       tcls:1;
  unsigned       cug :1;
  unsigned       cugo:1;
  unsigned       res1:1;
  unsigned       res2:1;
  unsigned       nui :1;
  unsigned           :21;
  unsigned char  recipient_tx_psiz;
  unsigned char  originator_tx_psiz;
  unsigned char  recipient_tx_wsiz;
  unsigned char  originator_tx_wsiz;
  unsigned char  recipient_tx_tcls;
  unsigned char  originator_tx_tcls;
  unsigned short reserved;
  unsigned short cug_index;
  unsigned short rpoa_id_count;
  unsigned short rpoa_id[30];
  unsigned int   nui_length;
  char           nui_data[109];
};
```

Fields in the Facilities Structure

The following fields are bits: (A value of 0 indicates false; a value of 1 indicates true.)

`facs`	Indicates whether there are *any* facilities being requested or not. If this field is set to 0, the whole of the remainder of the facilities structure is ignored.
`rpoa`	Indicates whether a recognized private operating agency is to be used.
`psiz`	Indicates whether a non-default packet size is to be used.
`wsiz`	Indicates whether a non-default window size is to be used.
`tcls`	Indicates whether a non-default throughput class is to be used.
`cug`	Indicates whether an index to a closed user group is to be supplied.
`cugo`	Indicates whether an index to a closed user group with outgoing access is to be supplied.
`res1`	Reserved.
`res2`	Reserved.
`nui`	Indicates whether network user identification is being supplied to the network.

The remaining fields provide the values or data associated with each of the above facilities bits that are set to 1. If the corresponding facilities bit is set to 0, each of these fields is ignored:

`recipient_tx_psiz`

The coded value of packet size to be used when sending data to the node that initiated the call. The values are coded as follows:

0x06	64 octets
0x07	128 octets
0x08	256 octets
0x09	512 octets
0x0A	1024 octets
0x0B	2048 octets
0x0C	4096 octets.

Note: The 4096-octet packets are allowed only in the 1984 CCITT recommendation and, for the call to be valid, the value of the X.25 CCITT attribute and the corresponding QLLC attribute must be set to 1984.

`originator_tx_psiz`

The coded value of packet size to be used when sending data from the node that initiated the call. The values are coded as for the `recipient_tx_psiz` field.

`recipient_tx_wsiz`

Reserved for QLLC use.

`originator_tx_wsiz`

Reserved for QLLC use.

`recipient_tx_tcls`

The coded values of throughput class requested for this virtual circuit, when sending data to the node that initiated the call. The values are coded as follows:

0x07	1200 bits per second
0x08	2400 bits per second

0x09	4800 bits per second
0x0A	9600 bits per second
0x0B	19200 bits per second
0x0C	48000 bits per second.

`originator_tx_tcls`
: The coded values of throughput class requested for this virtual circuit, when sending data from the node that initiated the call. The values are coded as for the `recipient_tx_tcls` field.

`cug_index`
: The decimal value of the index of the closed user group (CUG) within which this call is to be placed. This field is used for either CUG or CUG with outgoing access (CUGO) facilities.

`rpoa_id_count`
: The number of recognized private operating agency (RPOA) identifiers to be supplied in the `rpoa_id` field.

`rpoa_id`
: An array of RPOA identifiers that contains the number of identifiers specified in the `rpoa_id_count` field. The RPOA identifiers appear in the order in which they will be traversed during establishment of the call. The content of each array element is the decimal value of an RPOA identifier.

`nui_length`
: The length, in bytes, of the `nui_data` field.

`nui_data`
: Network user identification (NUI) data. The contents of this array are defined by you, in conjunction with the network provider. Note that the maximum allowable X.25 facilities string is 109 bytes. Even if NUI is the only facility requested, the facility code occupies one byte, so it is impossible to send more than 108 bytes of NUI data. Each additional facility requested reduces the space available for NUI data.

Asynchronous Function Subroutine Calls

Network and datagram data are not supported, so the **rcvn_fa** and **rcvd_fa** data functions are never called by DLCQLLC.DLCQLLC is compatible with each of the other asynchronous function subroutine calls for the kernel user.

Related Information

X.25 Routing Overview, X.25 Packet Switching Overview, List of X.25 Diagnostic Codes in *Communication Concepts and Procedures.*

The **cb_fac_struct** structure.

The **dlcclose** routine, **dlcconfig** routine, **dlcmpx** routine, **dlcopen** routine, **dlcread** routine, **dlcselect** routine, **dlcwrite** routine, **dlcioctl** routine.

The **ioctl** subroutine.

Token–Ring Data Link Control (DLCTOKEN) Overview on page 1–16, IEEE 802.3 Ethernet Data Link Control (DLC8023) Overview on page 1–26, Synchronous Data Link Control (DLCSDLC) Overview on page 1–44, Standard Ethernet Data Link Control (DLCETHER) Overview on page 1–35.

Chapter 2. Lists of DBM, NDBM, and NIS Subroutines

This chapter contains lists of subroutines for DBM, NDBM and NIS. These subroutines maintain key or content pairs in a database.

Alphabetical List of DBM Subroutines

The DBM subroutines maintain key/content pairs in a database. The DBM library has been superseded by the NDBM library and is now implemented with NDBM subroutines.

dbmclose
: Closes a database. The programmer must close one database before opening another database. The equivalent NDBM subroutine is the **dbm_close** subroutine.

dbminit
: Opens a database. The equivalent NDBM subroutine is the **dbm_open** subroutine.

delete
: Deletes a key and its associated contents. The equivalent NDBM subroutine is the **dbm_delete** subroutine.

fetch
: Accesses the data stored under a key. The equivalent NDBM subroutine is the **dbm_fetch** subroutine.

firstkey
: Makes a linear pass through all keys in the database and returns the first key that matches the specification. The equivalent NDBM subroutine is the **dbm_firstkey** subroutine.

nextkey
: Traverses the database and returns the next key in the database. The equivalent NDBM subroutine is the **dbm_nextkey** subroutine.

store
: Stores data under a key. The equivalent NDBM subroutine is the **dbm_store** subroutine.

Alphabetical List of NDBM Subroutines

The NDBM subroutines maintain key/content pairs in a data base. These routines handle large databases and access a keyed item in one or two file system accesses. The NDBM library replaces the earlier DBM library, which managed only a single database.

dbm_close	Closes a database.
dbm_delete	Deletes a key and its associated contents.
dbm_fetch	Accesses data stored under a key.
dbm_firstkey	Returns the first key in the database.
dbm_nextkey	Returns the next key in the database.
dbm_open	Opens a database for access.
dbm_store	Stores data under a key.

Alphabetical List of NIS Subroutines

The following is a list of the IBM NFS network information service subroutines:

yp_all	Transfers all of the key-value pairs from the network information service (NIS) server to the client as the entire map.
yp_bind	Calls the **ypbind** daemon directly for processes that use backup strategies when NIS is not available.
yperr_string	Returns a pointer to an error message string.
yp_first	Returns the first key-value pair from the named NIS map in the named domain.
yp_get_default_domain	Gets the default domain of the node.
yp_master	Returns the machine name of the NIS master server for a map.
yp_match	Searches for the value associated with a key.
yp_next	Returns each subsequent value it finds in the named NIS map until it reaches the end of the list.
yp_order	Returns the order number for an NIS map that identifies when the map was built.
ypprot_err	Takes an NIS protocol error code as input, and returns an error code to be used as input to a **yperr_string** subroutine.
yp_unbind	Manages socket descriptors for processes that access multiple domains.
yp_update	Makes changes to the NIS map.

Chapter 3. eXternal Data Representation (XDR)

The eXternal Data Representation (XDR) is a standard for the description and encoding of data. XDR uses a language to describe data formats, but the language is used only for describing data and is not a programming language. This chapter contains information on XDR criteria, implementing XDR, installing, solving XDR problems, and programming for XDR. The chapter is divided into concepts, lists of XDR subroutines, and examples.

eXternal Data Representation (XDR) Overview

The eXternal Data Representation (XDR) is a standard for the description and encoding of data. XDR uses a language to describe data formats, but the language is used only for describing data and is not a programming language. Protocols such as Remote Procedure Call (RPC) and the Network File System (NFS) use XDR to describe their data formats.

XDR not only solves data portability problems, it permits the reading and writing of arbitrary C language constructs in a consistent and well-documented manner. Therefore, it makes sense to use the XDR library routines even when the data is not shared among machines on a network.

The XDR standard does not depend on machine languages, manufacturers, operating systems, or architectures. This enables networked computers to share data regardless of the machine on which the data is produced or consumed. The XDR language permits transfer of data between different computer architectures and has been used to communicate data between such diverse machines as the VAX, IBM-PC, and Cray.

RPC uses XDR to establish uniform representations for data types in order to transfer message data between machines. For basic data types, such as integers and strings, XDR provides filter primitives that serialize, or translate, information from the local host's representation to XDR's representation. Likewise, XDR filter primitives deserialize XDR's data representation to the local host's data representation. XDR constructor primitives allow the use of the basic data types to create more complex data types such as arrays and discriminated unions.

The XDR routines that are called directly by RPC routines can be found in the Alphabetical List of RPC Subroutines and Macros on page 7–41. The XDR routines can be found in the Alphabetical List of XDR Subroutines and Macros on page 3–24.

A Canonical Standard

The XDR approach to standardizing data representations is *canonical*. That is, XDR defines a single byte (*big endian*), a single floating-point representation (IEEE), and so on. Any program running on any machine can use XDR to create portable data by translating its local representation to the XDR standards. Similarly, any program running on any machine can read portable data by translating the XDR standard representations to its local equivalents. The canonical standard decouples programs that create or send portable data from those that use or receive portable data.

The advent of a new machine or new language has no effect upon the community of existing portable data creators and users. A new machine can be programmed to convert both the standard representations and its local representations regardless of the local representations of other machines. Conversely, the local representations of the new machine are also irrelevant to existing programs running on other machines. These existing programs

can immediately read portable data produced by the new machine, because such data conforms to canonical standards.

There are strong precedents for XDR's canonical approach. All protocols below layer five of the ISO model (including TCP/IP, UDP/IP, XNS, and Ethernet) are canonical protocols. XDR fits into the ISO presentation layer and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference here is that XDR uses implicit typing, while X.409 uses explicit typing. With XDR, a single set of conversion routines need only be written once.

The time spent converting to and from a canonical representation is insignificant, especially in networking applications. When preparing a data structure for transfer, traversing the elements of the structure requires more time than converting the data. In networking applications, additional time is required to move the data down through the sender's protocol layers, across the network, and up through the receiver's protocol layers. Every machine must traverse and copy data structures, whether or not conversion is required.

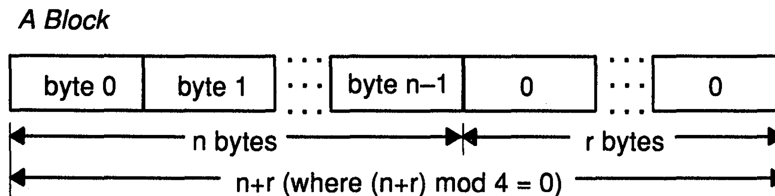
Basic Block Size

The XDR language is based on the assumption that bytes (8 bits of data, or an octet) can be ported to and encoded on media that preserve the meaning of the bytes across the hardware boundaries of data. XDR does not represent bit fields or bit maps. It represents data in blocks of multiples of 4 bytes (32 bits). The bytes are numbered from 0 (zero) to the value of $n - 1$, where the value (n mod 4) = 0. They are read from or written to a byte stream in order, such that byte m precedes byte $m + 1$.

Bytes are ported and encoded from low order to high order in local area networks. Representing data in standardized formats resolves situations that occur when different byte-ordering formats exist on networked machines. This also enables machines with different structure-alignment algorithms to communicate with each other.

See the following figure for an illustration of a block.

A Block



In a graphics box illustration, each box is delimited by a + (plus) sign at the 4 corners and vertical bars and dashes. Each box depicts a byte. The . . . (ellipses) between boxes show zero or more additional bytes where required.

Planned Enhancements

The XDR standard currently lacks representations for bit fields and bit maps because the standard is based on bytes. Packed, or binary-coded, decimals are also missing.

The XDR standard describes only the most commonly used data types of high-level languages, such as C or Pascal. This enables applications that are written in these languages to communicate easily over some medium.

Future extensions to XDR may permit the description of almost any existing protocol, such as TCP/IP. Support for different block sizes and byte orders is a minimum requirement for these protocols. With such support, XDR might be considered the 4-byte big endian member of a larger XDR family.

Related Information

Alphabetical List of XDR Subroutines and Macros on page 3–24, Alphabetical List of RPC Subroutines and Macros on page 7–41.

Functional List of XDR Subroutines and Macros on page 3–26, Functional List of RPC Subroutines and Macros on page 7–44.

Understanding the XDR Subroutine Format on page 3–3, Using the XDR Library on page 3–4, Understanding the XDR Language Specification on page 3–5, Understanding XDR Data Types on page 3–8, Understanding XDR Library Filter Primitives on page 3–17, Understanding XDR Non-Filter Primitives on page 3–20.

Remote Procedure Call (RPC) Overview for Programming on page 7–1.

Network File System Overview for System Management, Understanding Protocols for TCP/IP, User Datagram Protocol (UDP) in *Communication Concepts and Procedures.*

Understanding the XDR Subroutine Format

An XDR routine is associated with each data type. XDR routines have the following format:

```
xdr_XXX (XDRS, FP)
        XDR *XDRS;
        XXX *FP;
{
}
```

The parameters are described as follows:

XXX	An XDR data type.
XDRS	An opaque handle that points to an XDR stream. The opaque handle pointer is passed to the primitive XDR routines.
FP	An address of the data value that provides data to the stream or receives data from it.

The XDR routines usually return a value of 1 if successful and a value of 0 if unsuccessful. Return values other than these are noted within the description of the appropriate routine.

Related Information

Alphabetical List of RPC Subroutines and Macros on page 7–41, Alphabetical List of XDR Subroutines and Macros on page 3–24.

Functional List of RPC Subroutines and Macros on page 7–44, Functional List of XDR Subroutines and Macros on page 3–26.

Using the XDR Library on page 3–4, Understanding the XDR Language Specification on page 3–5, Understanding XDR Data Types on page 3–8, Understanding XDR Library Filter Primitives on page 3–17, Understanding XDR Non-Filter Primitives on page 3–20.

Using the XDR Library

The XDR library includes routines that permit programmers not only to read and write C language constructs, but also to write XDR routines that define other data types.

The XDR library includes the following:

- Library primitives for basic data types and constructed data types. The basic data types include number filters for integers, floating-point, and double-precision numbers, enumeration filters, and a routine for passing no data. Constructed data types include the filters for strings, arrays, unions, pointers, and opaque data.
- Data stream creation routines that call streams for serializing and deserializing data to or from standard I/O file streams, TCP/IP connections, and memory.
- Routines for the implementation of new XDR streams.
- Routines for passing linked lists.

See the Example Showing the Justification for Using XDR on page 3–32.

XDR with RPC

The XDR subroutines and macros may be called explicitly or by an RPC routine. When using XDR with RPC, clients do not create data streams. Instead, the RPC interface creates the streams. The RPC interface passes the information about a data stream as opaque data in the form of handles. This opaque data handle is referred to in routines as the *xdrs* parameter. Programmers who use C language programs with XDR routines must include the **<rpc/xdr.h>** header file, which contains the necessary XDR interfaces.

XDR Operation Directions

The XDR routines are not dependent on direction. The operation direction represented by `xdrs->xop` can have the *XDR_ENCODE*, *XDR_DECODE*, or *XDR_FREE* value. These operation values are handled internally by the XDR routines, which means the same XDR routine can be called to serialize or deserialize data. To achieve this independence, XDR passes the address of the object instead of passing the object itself.

Related Information

Alphabetical List of XDR Subroutines and Macros on page 3–24.

Understanding the XDR Subroutine Format on page 3–3, Understanding the XDR Language Specification on page 3–5, Understanding XDR Data Types on page 3–8, Understanding XDR Library Filter Primitives on page 3–17, Understanding XDR Non-Filter Primitives on page 3–20.

Example Showing the Justification for Using XDR on page 3–32.

Understanding Protocols for TCP/IP in *Communication Concepts and Procedures.*

Understanding the XDR Language Specification

The XDR language specification uses an extended Backus-Naur Form notation for describing the XDR language. The following is a brief description of the notation:

- The following characters are special characters:

  | A vertical bar separates alternative items.
  ( ) Parentheses enclose items that are grouped together.
  [ ] Brackets enclose optional items.
  , A comma separates more than one variable.
  * An asterisk following an item means 0 or more occurrences of the item.

- Terminal symbols are strings of special and non-special characters surrounded by " " (double quotes).
- Non-terminal symbols are strings of non-special characters.

The following specification illustrates the XDR notation:

```
"a" "very" ("," "very")* ["cold" "and"] "rainy" ("day" | "night")
```

An infinite number of strings match this pattern. Some of them are:

- `"a very rainy day"`
- `"a very, very rainy day"`
- `"a very cold and rainy day"`
- `"a very, very, very cold and rainy night"`

Lexical Notes

The following lexical notes apply to XDR language specification:

- Comments begin with a /* (backslash character followed by an asterisk) and terminate with a */ (asterisk followed by a backslash character).
- White space is used to separate items and is otherwise ignored.
- An identifier is a letter followed by an optional sequence of letters, digits, or an _ (underscore). Identifiers are case-sensitive.
- A constant is a sequence of one or more decimal digits, optionally preceded by a – (minus sign).

Declarations, Enumerations, Structures, and Unions

The XDR syntax describes declarations, enumerations, structures, and unions:

```
declaration:
  type–specifier identifier
  | type–specifier identifier "[" value "]"
  | type–specifier identifier "<" [ value ] "<"
  | "opaque" identifier "[" value "]"
  | "string" identifier "[" value "]"
  | type–specifier "*" identifier
  |"void"
```

```
value:
  constant
   | identifier

type-specifier:
    [ "unsigned" ] "int"
    | [ "unsigned" ] "hyper"
    | "float"
    | "double"
    | "bool"
    | enum-type-spec
    | struct-type-spec
    | union-type-spec
    | identifier

enum-type-spec:
  "enum" enum-body

enum-body:
  "{"
  ( identifier "=" value )
  ("," identifier "=" value )*
  "}"

struct-type-spec:
  "struct" struct-body

struct-body:
  "{"
  ( declaration ";" )
  ( declaration ";" )*
  "}"

union-type-spec:
  "union" union-body

union-body:
  "switch" "(" declaration ")" "{"
  ( "case" value ":" declaration ";" )
  ( "case" value ":" declaration ";" )*
  [ "default" ":" declaration ";" ]
  "}"

constant-def:
  "const" identifier "=" constant ";"

type-def
```

```
  "typedef" declaration ";"
  | "enum" identifier enum-body ";"
  | "struct" identifier struct-body ";"
  | "union" identifier union-body ";"

definition:
  type-def
  | constant-def

specification:
  definition *
```

Syntax Notes

Following are some additional notes pertaining to the XDR language syntax.

1. The following keywords cannot be used as identifiers:
   - **bool**
   - **case**
   - **const**
   - **default**
   - **double**
   - **enum**
   - **float**
   - **hyper**
   - **opaque**
   - **string**
   - **struct**
   - **switch**
   - **typedef**
   - **union**
   - **unsigned**
   - **void**
2. Only unsigned constants can be used as size specifications for arrays. If an identifier is used, it must be declared previously as an unsigned constant in a `const` definition.
3. Constant and type identifiers within the scope of a specification are in the same name space and must be declared uniquely within this scope.
4. Variable names must be unique within the scope of `struct` and `union` declarations. Nested `struct` and `union` declarations create new scopes.

Related Information

Understanding the XDR Subroutine Format on page 3–3, Using the XDR Library on page 3–4, Understanding XDR Data Types on page 3–8, Understanding XDR Library Filter Primitives on page 3–17, Understanding XDR Non-Filter Primitives on page 3–20.

Understanding XDR Data Types

The following basic and constructed data types are defined in the XDR standard:

- Integers
- Enumerations
- Booleans
- Floating-point decimals
- Opaque data
- Arrays
- String
- Structure
- Discriminated union
- Void
- Constant
- Typedef
- Optional data.

A general paradigm declaration is shown for each type. The < and > (angle brackets) denote variable-length sequences of data, while the [and] (square brackets) denote fixed-length sequences of data. The letters `n`, `m`, and `r` denote integers. See the Example Using an XDR Data Description on page 3–40 for an extensive example of the data types.

Integer Data Types

XDR defines two integer data types: signed and unsigned integer and hyper-integer.

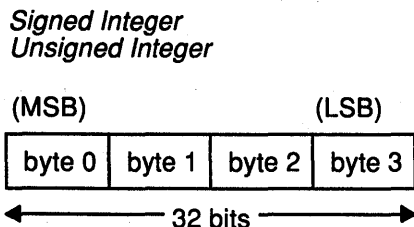
Signed and Unsigned Integers

The XDR standard defines signed integers as *integer*. A signed integer is a 32-bit datum that encodes an integer in the range [–2147483648 to 2147483647]. The signed integer is represented in two's complement notation. The most significant byte is 0; the least significant byte is 3.

An unsigned integer is a 32-bit datum that encodes a nonnegative integer in the range [0 to 4294967295]. The unsigned integer is represented by an unsigned binary number whose most significant byte is 0; the least significant byte is 3.

See the following figure for an illustration of signed and unsigned integers.

Signed Integer
Unsigned Integer

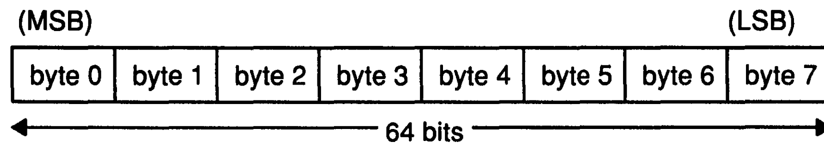


Signed and Unsigned Hyper-Integers

The XDR standard also defines 64-bit (8-byte) numbers called hyper-integers and unsigned hyper-integers. Their representations are the extensions of integers and unsigned integers. Hyper-integers are represented in two's complement notation. The most significant byte is 0; the least significant byte is 7.

See the following figure for an illustration of signed and unsigned hyper integer.

Signed Hyper-Integer
Unsigned Hyper-Integer



Enumeration Data Type

The XDR standard provides enumerations for describing subsets of integers. XDR defines enumerations as *enum.* Enumerations have the same representation as signed integers and are declared as follows:

```
enum { name–identifier = constant, ... } identifier;
```

Encoding as `enum` any integers other than those with assignments in the `enum` declaration causes an error condition.

Boolean Data Type

Booleans occur frequently enough to warrant an explicit type in the XDR standard.

Booleans are declared as follows:

```
bool identifier;
```

This declaration is equivalent to:

```
enum { FALSE = 0, TRUE = 1 } identifier;
```

Floating-Point Data Type

The XDR standard defines two floating-point data types: single-precision and double-precision floating points.

Single-Precision Floating–Point

XDR defines the single-precision floating-point data type as a *float.* The length of a float is 32 bits, or 4 bytes. Floats are encoded using the IEEE standard for normalized single-precision floating-point numbers.

The single-precision floating-point number is described as follows:

```
(–1)**S * 2**(E–Bias) * 1.F
```

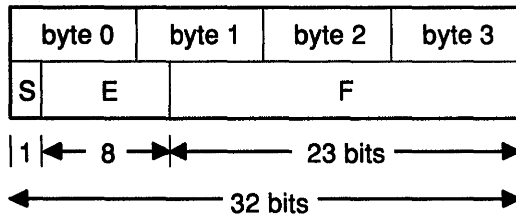
`S` Sign of the number. This is a 1-bit field that contains the value 0 to represent positive or 1 to represent negative.

`E` Exponent of the number in base 2. This field contains 8 bits. The exponent is biased by 127.

`F` Fractional part of the number's mantissa in base 2. This field contains 23 bits.

See the following figure for an illustration of the single-precision floating-point data type.

Single–Precision Floating–Point



The most and least significant bits of a single-precision floating-point number are 0 and 31. The beginning (and most significant) bit offsets of S, E, and F are 0, 1, and 9, respectively. These numbers refer to the mathematical positions of the bits and *not* to their physical locations, which vary from medium to medium.

The IEEE specifications should be considered for the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). According to IEEE specifications, the NaN (not-a-number) is system-dependent and should not be used externally.

Double-Precision Floating–Point

The XDR standard defines the encoding for the double-precision floating-point data type as a *double*. The length of a double is 64 bits or 8 bytes. Doubles are encoded using the IEEE standard for normalized double-precision floating-point numbers.

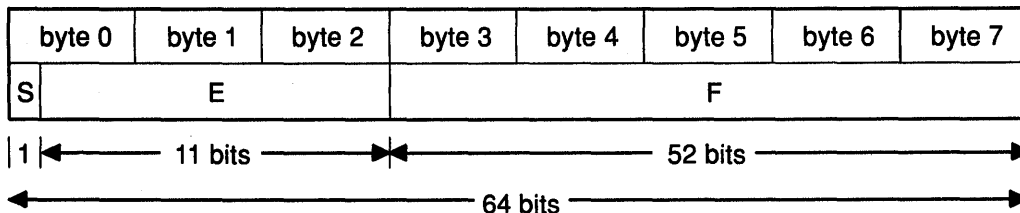
The double-precision floating-point data type is described by:

```
(-1)**S * 2**(E-Bias) * 1.F
```

S	Sign of the number. This 1-bit field contains the value 0 to represent positive or 1 to represent negative.
E	Exponent of the number, base 2. This field contains 11 bits. The exponent is biased by 1023.
F	Fractional part of the number's mantissa in base 2. This field contains 52 bits.

See the following figure for an illustration of the double-precision floating-point data type.

Double–Precision Floating–Point



The most and least significant bits of a double-precision floating-point number are 0 and 63. The beginning (and most significant) bit offsets of S, E, and F are 0, 1, and 12, respectively. These numbers refer to the mathematical positions of the bits and *not* to their physical locations, which vary from medium to medium.

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). According to IEEE specifications, the NaN (not-a-number) is system-dependent and should not be used externally.

Opaque Data Type

The XDR standard defines two types of opaque data: fixed-length and variable-length opaque data.

Fixed–Length Opaque Data

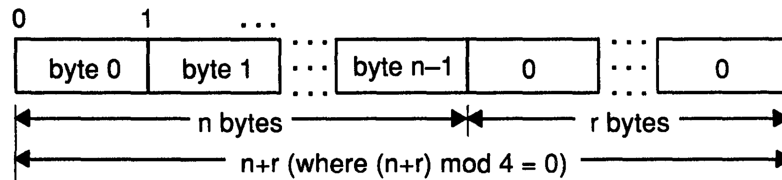
XDR defines fixed-length uninterpreted data as *opaque*. Fixed-length opaque data is declared as follows:

```
opaque identifier[n];
```

The constant n is the (static) number of bytes necessary to contain the opaque data. If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count of the opaque object a multiple of four.

See the following figure for an illustration of the fixed-length opaque data type.

Fixed–Length Opaque



Variable–length Opaque Data

The XDR standard also provides for variable-length (counted) opaque data which is defined as a sequence of n arbitrary bytes and numbered 0 through n–1. Opaque data is encoded as an unsigned integer and followed by the n bytes of the sequence.

Byte m of the sequence always precedes byte m+1, and byte 0 of the sequence always follows the sequence length (count). Enough (0 to 3) residual zero bytes, r, are added to make the total byte count a multiple of four.

Variable-length opaque data is declared in one of the following forms:

```
opaque identifier<m>;
```

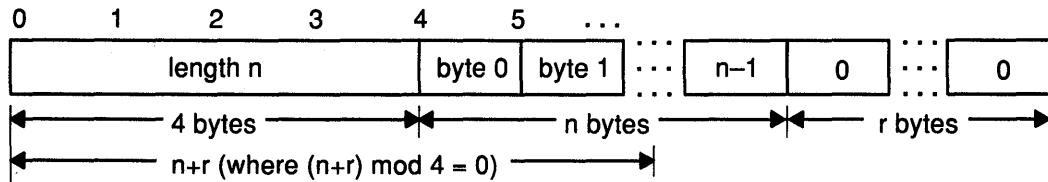
OR

```
opaque identifier<>;
```

The constant m denotes an upper bound for the number of bytes that the sequence can contain. If m is not specified, as in the second declaration, it is assumed to be (2**32) – 1, which is the maximum length. The constant m would normally be found in a protocol specification.

See the following figure for an illustration of the variable-length opaque data type.

Variable–Length Opaque



Encoding a length that is greater than the maximum described in the specification is an error.

Array Data Type

The XDR standard defines two type of arrays: fixed-length or variable-length arrays.

Fixed–Length Array

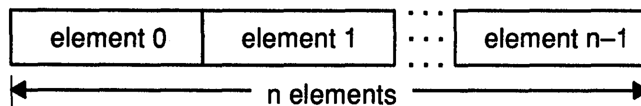
Fixed-length arrays of homogeneous elements are described in the following form:

```
type–name identifier[n];
```

Fixed-length arrays of elements are encoded by individually coding the elements of the array in their natural order, `0` through `n–1`. Each element size is a multiple of four bytes. Although the elements are of the same type, they may have different sizes. For example, in a fixed-length array of strings, all elements are of the string type, yet each element varies in length.

See the following figure for an illustration of fixed-length arrays.

Fixed–Length Array



Variable–Length Array

The XDR standard provides counted byte arrays for encoding variable-length arrays of homogeneous elements. The array is encoded as the element count `n` (an unsigned integer) followed by the encoding of each of the array's elements, starting with element 0 and progressing through element `n–1`.

Variable-length arrays are described as follows:

```
type–name identifier<m>;
```

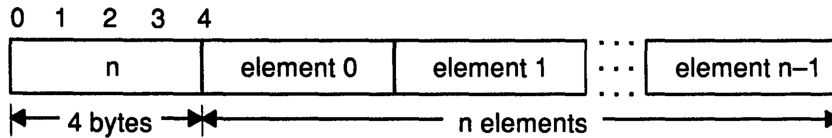
OR

```
type–name identifier<>;
```

The constant `m` specifies the maximum acceptable element count of an array. If `m` is not specified, it is assumed to be (2**32) – 1.

See the following figure for an illustration of variable-length arrays.

Variable–Length Array



Encoding a value of n that is greater than the maximum described in the specification is an error.

Strings

The XDR standard defines a string of n (numbered 0 through n–1) ASCII bytes to be the number n encoded as an unsigned integer and followed by the n bytes of the string. Byte m of the string always precedes byte m+1, and byte 0 of the string always follows the string length. If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four.

Counted byte strings are declared as one of the following:

```
string object<m>;
```

OR

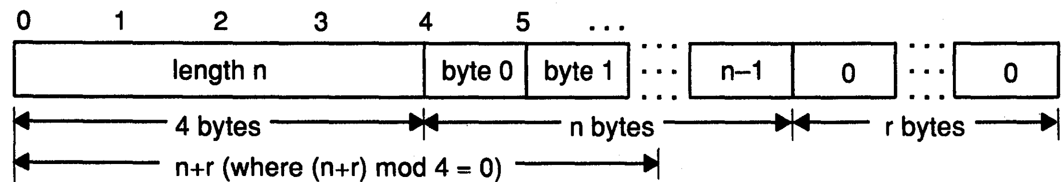
```
string object<>;
```

The constant m denotes an upper bound of the number of bytes that a string may contain. If m is not specified, as in the second declaration, it is assumed to be (2**32) – 1, which is the maximum length. The constant m would normally be found in a protocol specification. For example, a filing protocol may state that a file name can be no longer than 255 bytes, as follows:

```
string filename<255>;
```

See the following figure for an illustration of counted byte strings.

Counted Byte String



Encoding a length greater than the maximum described in the specification causes an error condition.

Structures

Using the the primitive routines, the programmer can write unique XDR routines to describe arbitrary data structures such as elements of arrays, arms of unions, or objects pointed to from other structures. The structures themselves may contain arrays of arbitrary elements or pointers to other structures.

Structures are declared as follows:

```
struct {
    component-declaration-A;
    component-declaration-B;
    ...
} identifier;
```

In a structure, the components are encoded in the order of their declaration in the structure. Each component's size is a multiple of four bytes, though the components may have different sizes.

See the following figure for an illustration of a structure.

Structure

Discriminated Unions

A discriminated union is a union data structure that holds various objects, with one of the objects identified directly by a discriminant, or arm. The discriminant is the first item to be serialized or deserialized. A discriminated union includes a discriminant and a component selected from a set of types that are prearranged according to the value of the discriminant. The type of discriminant is either integer, unsigned integer, or an enumerated type, such as `bool`. The component types are called *arms* of the union. The arms of a discriminated union are preceded by the value of the discriminant that implies their encoding. See the Example Using an XDR Discriminated Union on page 3-42.

Discriminated unions are declared as follows:

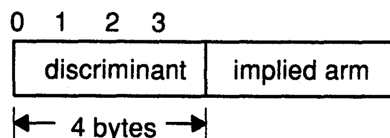
```
union switch (discriminant-declaration) {
    case discriminant-value-A:
    arm-declaration-A;
    case discriminant-value-B:
    arm-declaration-B;
    ...
    default: default-declaration;
} identifier;
```

Each `case` keyword is followed by a legal value of the discriminant. The default arm is optional. If an arm is not specified, a valid encoding of the union cannot take on unspecified discriminant values. The size of the implied arm is always a multiple of four bytes.

The discriminated union is encoded as the discriminant followed by the encoding of the implied arm.

See the following figure for an illustration of a discriminated union.

Discriminated Union



Voids

An XDR void is a zero-byte quantity. Voids are used for describing operations that take no data as input or output. Voids are also useful in unions, where some arms contain data and others do not.

The declaration for a void follows:

```
void;
```

Voids are illustrated as follows:

```
++
||
++
--><-- 0 bytes
```

Constants

A constant is used to define a symbolic name for a constant, and it does not declare any data. The symbolic constant may be used anywhere a regular constant may be used.

The data declaration for a constant follows this form:

```
const name-identifier = n;
```

The following example defines a symbolic constant `DOZEN` that is equal to `12`:

```
const DOZEN = 12;
```

Type Definitions

A type definition (**typedef**) does not declare any data but serves to define new identifiers for declaring data.

The syntax for a **typedef** is:

```
typedef declaration;
```

The new type name is the variable name in the declaration part of the **typedef**. For example, the following defines a new type called `eggbox`, using an existing type called `egg`:

```
typedef egg eggbox[DOZEN];
```

Variables declared using the new type name have the same type as the new type name would have in the **typedef** if the type was considered a variable. For example, the following two declarations are equivalent in declaring the variable `fresheggs`:

```
eggbox fresheggs;
egg    fresheggs[DOZEN];
```

A **typedef** has the following form:

```
typedef <<struct, union, or enum definition>> identifier;
```

There is an alternative **typedef** form that is preferred for structures, unions, and enumerations. The **typedef** form can be converted to the alternative form by removing `typedef` and placing the identifier after the `struct`, `union`, or `enum` keyword, instead of at the end. For example, here are the two ways to define the type `bool`:

```
enum bool {        /*  preferred alternative  */
    FALSE = 0,
    TRUE = 1
    };
```

This syntax is preferred because the programmer does not have to wait until the end of a declaration to determine the name of the new type.

Optional Data

Optional data is a type of union that occurs so frequently that it has its own syntax. The optional data type has a close correlation to the representation of recursive data structures by use of pointers in high-level languages, such as C or Pascal. The syntax for pointers is the same as that for C language.

The syntax for optional data is as follows:

```
type–name *identifier;
```

The declaration for optional data is equivalent to the following union:

```
union switch (bool opted) {
    case TRUE:
    type–name element;
    case FALSE:
    void;
} identifier;
```

Since the boolean opted can be interpreted as the length of the array, the declaration for optional data is also equivalent to the following variable-length array declaration:

```
type–name identifier<1>;
```

Optional data is very useful for describing recursive data structures such as linked lists and trees. For example, the following defines a `stringlist` type that encodes lists of arbitrary length strings:

```
struct *stringlist {
    string item<>;
    stringlist next;
};
```

The previous example can be equivalently declared as a union, as follows:

```
union stringlist switch (bool opted) {
    case TRUE:
        struct {
            string item<>;
            stringlist next;
        } element;
    case FALSE:
        void;
};
```

The same example can also be declared as a variable-length array, as follows:

```
struct stringlist<1> {
    string item<>;
    stringlist next;
};
```

Since both the union and the array declarations obscure the intention of the `stringlist` type, the optional data declaration is preferred.

Related Information

Understanding the XDR Language Specification on page 3–5, Understanding the XDR Subroutine Format on page 3–3, Using the XDR Library on page 3–4, Understanding XDR Data Types on page 3–8, Understanding XDR Library Filter Primitives on page 3–17, Understanding XDR Non–Filter Primitives on page 3–20.

Understanding XDR Library Filter Primitives

The XDR primitives are routines that define the basic and constructed data types. The XDR language provides programmers with a specification for uniform representations that includes filter primitives for basic and constructed data types. The basic data types include integers, enumerations, Booleans, hyper-integers, floating points, and void data. The constructed data types include: strings, structures, byte arrays, arrays, opaque data, unions, and pointers.

The XDR standard translates both basic and constructed data types. For basic data types such as integers and strings, XDR provides basic filter primitives that serialize information from the local host's representation to XDR representation, and deserialize information from the XDR representation to the local host's representation. For constructed data types, XDR provides constructed filter primitives that allow the use of basic data types, such as integers and floating-point numbers, to create more complex constructs such as arrays and discriminated unions.

RPC uses XDR to establish uniform representations for data types in order to transfer the call message data between machines. Although the XDR constructs resemble the C programming language, it is important to note that C-language constructs define the code for programs, while XDR standardizes the representation of data types within the programming code itself.

Using XDR Basic Filter Primitives

The XDR primitives are routines that define the basic and constructed data types. The basic data type filter primitives include the following:

- Number filter primitives
- Floating-point filter primitives
- Enumeration filter primitives
- No data filter primitives.

Number Filter Primitives

The XDR library provides basic filter primitives that translate between types of numbers and their external representations. The XDR number filters cover signed and unsigned integers, as well as signed and unsigned short and long integers.

The routines for the XDR number filters are:

xdr_int Translates between C language integers and their external representations.
xdr_u_int Translates between C language unsigned integers and their external representations.
xdr_long Translates between C language long integers and their external representations.
xdr_u_long Translates between C language unsigned long integers and their external representations.
xdr_short Translates between C language short integers and their external representations.
xdr_u_short Translates between C language unsigned short integers and their external representations.

Floating–Point Filter Primitives

The XDR library provides primitives that translate between floating-point data and their external representations. Floating-point data encodes an integer with an exponent. Floats and double-precision numbers compose floating-point data.

Note: Numbers are represented as IEEE standard floating points. Routines may fail when decoding IEEE representations into machine-specific representations, or vice versa.

The routines for the XDR floating-point filters are:

xdr_float Translates between C language floats and their external representations.
xdr_double Translates between C language double-precision numbers and their external representations.

Enumeration Filter Primitives

The XDR library provides a primitive for generic enumerations based on the assumption that a C enumeration value (**enum**) has the same representation. There is a special enumeration in XDR known as the *Boolean*.

The routines for the XDR library enumeration filters are the following:

xdr_enum Translates between C language enums and their external representations.
xdr_bool Translates between Booleans and their external representations.

Passing No Data

Sometimes an XDR routine must be supplied to the RPC system, but no data is required or passed. The XDR library provides the following primitive for this function:

xdr_void Supplies an XDR subroutine to the RPC system without transmitting data.

Using XDR Constructed Filter Primitives

The XDR filter primitives are routines that define the basic and constructed data types. Constructed data type filters allow complex data types to be created from the basic data types. Constructed data types require more parameters to perform more complicated functions than the basic data types. Memory management is an example of a more complicated function that can be performed with the constructed primitives. Memory is allocated when deserializing data with the **xdr_decode** routine. Memory is deallocated through the **xdr_free** routine.

The constructed data type filter primitives include the following:

- String filter primitives
- Array filter primitives
- Opaque data filter primitives
- Pointers to structures
- Discriminated unions.

String Filter Primitives

A string is a constructed filter primitive that consists of a sequence of bytes terminated by a null byte. The null byte does not figure into the length of the string. Externally, strings are represented by a sequence of ASCII characters. Internally, XDR represents them as pointers to characters with the designation `char *`.

The XDR library includes primitives for the following string routines:

xdr_string Translates between C language strings and their external representations.

xdr_wrapstring Calls the **xdr_string** subroutine.

Array Filter Primitives

Arrays are constructed filter primitives and may be generic arrays or byte arrays. The XDR library provides filter primitives for handling both types of arrays.

Generic Arrays

Generic arrays consist of arbitrary elements. Generic arrays are handled in much the same way as byte arrays, which handle a subset of generic arrays where the size of the elements is 1 and their external descriptions are predetermined. The primitive for generic arrays requires an additional parameter to define the size of the element in the array and to call an XDR routine to encode or decode each element in the array.

The XDR library has the following routine for generic arrays:

xdr_array Translates between variable-length arrays and their corresponding external representations.

xdr_vector Translates between fixed-length arrays and their corresponding external representations.

Byte Arrays

The XDR library provides a primitive for byte arrays. Although similar to strings, byte arrays differ from strings by having a byte count. That is, the length of the array is set by an unsigned integer. They also differ in that byte arrays are not terminated with a null character. External and internal representations of byte arrays are the same.

The XDR library includes the following routine for byte arrays:

xdr_bytes Translates between counted byte string arrays and their external representations.

Opaque Data Filter Primitives

Opaque data is composed of bytes of a fixed size that are not interpreted as they pass through the data streams. Opaque data bytes, such as handles, are passed between server and client without being inspected by the client. The client uses the data as it is and then returns it to the server. By definition, the actual data contained in the opaque object is not portable between computers.

The XDR library includes the following routine for opaque data:

xdr_opaque Translates between opaque data and its external representation.

Primitive for Pointers to Structures

The XDR library provides the primitive for pointers so that structures referenced within other structures can be easily serialized, deserialized, and freed.

The XDR library includes the following routine for pointers to structures:

xdr_reference Provides pointer chasing within structures.

Primitive for Discriminated Unions

A discriminated union is a C language union, which is an object that holds several data types with one arm of the union an enumeration value, or discriminant, that holds a specific object to be processed over the system first. The discriminant is an enumeration value (**enum_t**).

The XDR library includes the following routine for discriminated unions:

xdr_union Translates between discriminated unions and their external representations.

Related Information

Alphabetical List of XDR Subroutines and Macros on page 3–24.

List of XDR Examples on page 3–28.

Understanding the XDR Subroutine Format on page 3–3, Using the XDR Library on page 3–4, Understanding the XDR Language Specification on page 3–5, Understanding XDR Data Types on page 3–8, Understanding XDR Non–Filter Primitives on page 3–20.

Understanding XDR Non–Filter Primitives

The XDR non-filter primitives are used to create, manipulate, implement, and destroy XDR data streams. These primitives allow the programmer to describe the data stream position, change the data stream position, and destroy a data stream.

Creating and Using XDR Data Streams

XDR data streams are obtained by calling creation routines that take arguments specifically designed to the properties of the stream. There are existing XDR data streams for serializing or deserializing data in standard input and output streams, memory streams, and record streams.

Note: RPC clients do not have to create XDR streams because the RPC system creates and passes these streams to the client.

The types of data streams include standard I/O streams, memory streams, and record streams.

Standard I/O Streams

XDR data streams serialize and deserialize standard input/output by calling the standard input/output creation routine to initialize the XDR data stream pointed to by the *xdrs* parameter.

The XDR library includes the following routine for standard I/O data streams:

xdrstdio_create Initializes the XDR data stream pointed to by the *xdrs* parameter.

Memory Streams

XDR data streams serialize and deserialize data from memory by calling the XDR memory creation routine to initialize in local memory the XDR stream pointed at by the *xdrs* parameter. In RPC, UDP/IP implementation of remote procedure calls uses this routine to build entire call and reply messages in memory before sending the message to the recipient.

The XDR library includes the following routine for memory data streams:

xdrmem_create Initializes in local memory the XDR stream pointed to by the *xdrs* parameter.

Record Streams

Record streams are XDR streams built on top of record fragments, which are built on TCP/IP streams. TCP/IP is a connection protocol for transporting large streams of data at one time instead of transporting a single data packet at a time.

The primary use of a record stream is to interface remote procedure calls to TCP connections. It can also be used to stream data into or out of normal files.

XDR provides the following routines for use with record streams:

xdrrec_create Provides an XDR stream that can contain long sequences of records.

xdrrec_endofrecord Causes the current outgoing data to be marked as a record.

xdrrec_skiprecord Causes the position of an input stream to move to the beginning of the next record.

xdrrec_eof Checks the buffer for an input stream that identifies the end of file (EOF).

Manipulating an XDR Data Stream

XDR provides routines for describing the data stream position and changing the data stream position:

xdr_getpos Returns an unsigned integer that describes the current position in the data stream.

xdr_setpos Changes the current position in the XDR stream.

Implementing an XDR Data Stream

XDR data streams can be created and implemented by programmers. The abstract data types (XDR handle) required for programmers to implement their own XDR streams are shown in the following example. They contain operations that are being applied to the stream, an operation vector for the particular implementation, and two private fields for the use of the particular implementation.

```
enum xdr_op { XDR_ENCODE=0, XDR_DECODE=1, XDR_FREE=2 };
typedef struct {
       enum xdr_op x_op;
       struct xdr_ops {
           bool_t   (*x_getlong)   ();
           boot_t   (*x_putlong)   ();
           boot_t   (*x_getbytes)  ();
           boot_t   (*x_putbytes)  ();
           u_int    (*x_getpostn)  ();
           boot_t   (*x_setpostn)  ();
           caddr_t  (*x_inline)    ();
           VOID     (*x_destroy)   ();

       } *XOp;
       caddr_t x_public;
       caddr_t x_private;
       caddr_t x_base;
       int     x_handy;
} XDR;
```

The following parameters are pointers to XDR stream manipulation routines:

`x_getlong`	Gets long integer values from the data stream.
`x_putlong`	Puts long integer values into the data stream.
`x_getbytes`	Gets bytes from the data stream.
`x_putbytes`	Puts bytes into the data stream.
`x_getpostn`	Returns stream offset.
`x_setpostn`	Repositions offset.
`x_inline`	Points to internal data buffer, which can be used for any purpose.
`x_destroy`	Frees private data structure.
XOp	Specifies the current operation being performed on the stream. This field is important to the XDR primitives. However, the stream's implementation does not depend on the value of this parameter.

The following fields are specific to a stream's implementation:

x_public	Specifies user data that is private to the stream's implementation and is not used by the XDR primitive.
x_private	Points to the private data.
x_base	Contains the position information in the data stream that is private to the user implementation.
x_handy	Data can contain extra information as necessary.

Destroying an XDR Data Stream

XDR provides a routine that destroys the XDR stream pointed to by the *xdrs* parameter and frees the private data structures allocated to the stream.

xdr_destroy
Destroys the XDR stream pointed to by the *xdrs* parameter.

The use of the XDR stream handle is undefined after it is destroyed.

Related Information

The **xdr_destroy** subroutine, **xdr_getpos** subroutine, **xdr_setpos** subroutine.

Alphabetical List of XDR Subroutines and Macros on page 3–24.

Understanding the XDR Subroutine Format on page 3–3, Using the XDR Library on page 3–4, Understanding the XDR Language Specification on page 3–5, Understanding XDR Data Types on page 3–8, Understanding XDR Library Filter Primitives on page 3–17.

Understanding Protocols for TCP/IP, User Datagram Protocol (UDP) in *Communication Concepts and Procedures.*

Alphabetical List of XDR Subroutines and Macros

xdr_array	Translates between variable-length arrays and their corresponding external representations.
xdr_bool	Translates between Booleans and their external representations.
xdr_bytes	Translates between internal counted byte string arrays and their external representations.
xdr_char	Translates between C language characters and their external representations.
xdr_destroy	Destroys the XDR stream pointed to by the *xdrs* parameter.
xdr_double	Translates between C language double-precision numbers and their external representations.
xdr_enum	Translates between C language enums and their external representations.
xdr_float	Translates between C language floats and their external representations.
xdr_free	Deallocates or frees memory.
xdr_getpos	Returns an unsigned integer that describes the current position in the data stream.
xdr_inline	Returns a pointer to the buffer of a stream pointed to by the *xdrs* parameter.
xdr_int	Translates between C language integers and their external representations.
xdr_long	Translates between C language long integers and their external representations.
xdr_opaque	Translates between fixed-length opaque data and its external representation.
xdr_pointer	Provides pointer chasing within structures and serializes NULL pointers.
xdr_reference	Provides pointer chasing within structures.
xdr_setpos	Changes the current position in the XDR stream.
xdr_short	Translates between C language short integers and their external representations.
xdr_string	Translates between C language strings and their external representations.
xdr_u_char	Translates between unsigned C language characters and their external representations.

xdr_u_int	Translates between C language unsigned integers and their external representations.
xdr_u_long	Translates between C language unsigned long integers and their external representations.
xdr_u_short	Translates between C language unsigned short integers and their external representations.
xdr_union	Translates between discriminated unions and their external representations.
xdr_vector	Translates between fixed-length arrays and their corresponding external representations.
xdr_void	Supplies an XDR subroutine to the RPC system without transmitting data.
xdr_wrapstring	Calls the **xdr_string** subroutine.
xdrmem_create	Initializes in local memory the XDR stream pointed to by the *xdrs* parameter.
xdrrec_create	Provides an XDR stream that can contain long sequences of records.
xdrrec_endofrecord	Causes the current outgoing data to be marked as a record.
xdrrec_eof	Checks the buffer for an input stream that identifies the end of file (EOF).
xdrrec_skiprecord	Causes the position of an input stream to move to the beginning of the next record.
xdrstdio_create	Initializes the XDR data stream pointed to by the *xdrs* parameter.

Related Information

Functional List of XDR Subroutines and Macros on page 3–26.

Understanding XDR Data Types on page 3–8, Understanding the XDR Subroutine Format on page 3–3, Understanding the XDR Language Specification on page 3–5, Using the XDR Library on page 3–4, Understanding XDR Library Filter Primitives on page 3–17, Understanding XDR Non-Filter Primitives on page 3–20.

Functional List of XDR Subroutines and Macros

The XDR library provides subroutines and macros that translate the C language to standardize data transport. The XDR library consists of filter primitives and non-filter primitives.

Using the XDR Library Filter Primitives

The XDR library provides the following filter primitives:

xdr_array Translates between variable-length arrays and their corresponding external representations.

xdr_bool Translates between Booleans and their external representations.

xdr_bytes Translates between internal counted byte string arrays and their external representations.

xdr_char Translates between C language characters and their external representations.

xdr_double Translates between C language double-precision numbers and their external representations.

xdr_enum Translates between C language enums and their external representations.

xdr_float Translates between C language floats and their external representations.

xdr_int Translates between C language integers and their external representations.

xdr_long Translates between C language long integers and their external representations.

xdr_opaque Translates between opaque data and its external representation.

xdr_reference Provides pointer chasing within structures.

xdr_short Translates between C language short integers and their external representations.

xdr_string Translates between C language strings and their external representations.

xdr_u_char Translates between unsigned C language characters and their external representations.

xdr_u_int Translates between C language unsigned integers and their external representations.

xdr_u_long Translates between C language unsigned long integers and their external representations.

xdr_u_short Translates between C language unsigned short integers and their external representations.

xdr_union Translates between discriminated unions and their external representations.

xdr_vector Translates between fixed-length arrays and their corresponding external representations.

xdr_void Supplies an XDR subroutine to the RPC system without transmitting data.

xdr_wrapstring Calls the **xdr_string** subroutine.

Using the XDR Library Non–Filter Primitives

The XDR library provides the following filter primitives:

xdr_destroy Destroys the XDR stream pointed to by the *xdrs* parameter.

xdr_getpos Returns an unsigned integer that describes the current position in the data stream.

xdr_setpos Changes the current position in the XDR stream.

xdr_inline Returns a pointer to an internal piece of the buffer of a stream, pointed to by the *xdrs* parameter.

xdrmem_create Initializes in local memory the XDR stream pointed to by the *xdrs* parameter.

xdr_pointer Provides pointer chasing within structures and serializes NULL pointers.

xdrstdio_create Initializes the XDR data stream pointed to by the *xdrs* parameter.

xdrrec_create Provides an XDR stream that can contain long sequences of records.

xdrrec_endofrecord Causes the current outgoing data to be marked as a record.

xdrrec_eof Checks the buffer for an input stream.

xdrrec_skiprecord Causes the position of an input stream to move to the beginning of the next record.

xdr_free Deallocates or frees memory.

Related Information

Alphabetical List of XDR Subroutines and Macros on page 3–24.

Understanding XDR Data Types on page 3–8, Understanding the XDR Subroutine Format on page 3–3, Understanding the XDR Language Specification on page 3–5, Using the XDR Library on page 3–4, Understanding XDR Library Filter Primitives on page 3–17, Understanding XDR Non-Filter Primitives on page 3–20.

List of XDR Examples

The following is a list of XDR examples:

Example Passing Linked Lists Using XDR on page 3–29

Example Showing the Justification for Using XDR on page 3–32

Example Showing the Use of Pointers in XDR on page 3–35

Example Using an XDRon page 3–36.

Example Using an XDR Array on page 3–37

Example Using an XDR Data Description on page3–40

Example Using an XDR Discriminated Union on page 3–42.

Related Information

Alphabetical List of XDR Subroutines and Macros on page 3–24, Functional List of XDR Subroutines and Macros on page 3–26.

Understanding XDR Data Types on page 3–8, Understanding the XDR Subroutine Format on page 3–3, Understanding the XDR Language Specification on page 3–5, Using the XDR Library on page 3–4, Understanding XDR Library Filter Primitives on page 3–17, Understanding XDR Non-Filter Primitives on page 3–20.

Example Passing Linked Lists Using XDR

Linked lists of arbitrary length can be passed using XDR. To help illustrate the functions of the XDR routine for encoding, decoding, or freeing linked lists, the following example creates a data structure and defines its associated XDR routine:

The Example Showing the Use of Pointers in XDR on page 3–35 presents a C data structure and its associated XDR routines for an individual's gross assets and liabilities. The example is duplicated below:

```
struct gnumbers {
        long g_assets;
        long g_liabilities;
};
bool_t
xdr_gnumbers (xdrs, gp)
        XDR *xdrs;
        struct gnumbers *gp;
{
        if (xdr_long (xdrs, &(gp->g_assets)))
            return (xdr_long (xdrs, &( gp->g_liabilities)));
        return(FALSE);
}
```

xdrs Points to the XDR data stream handle.

gp Points to the address of the structure that provides the data to or from the XDR stream.

To implement a linked list of such information, a data structure could be constructed as follows:

```
struct gnumbers_node {
        struct gnumbers gn_numbers;
        struct gnnumbers_node *gn_next;
};
typedef struct gnumbers_node *gnumbers_list;
```

The head of the linked list can be thought of as the data object; that is, the head is not merely a convenient shorthand for a structure. Similarly, the `gn_next` field is used to indicate whether or not the object has terminated. If the object continues, the `gn_next` field is also the address of where it continues. The link addresses carry no useful information when the object is serialized.

The XDR data description of this linked list can be described by the recursive declaration of the `gnumbers_list`, as follows:

```
struct gnumbers {
        int g_assets;
        int g_liabilities;
};
struct gnumbers_node {
        gnumbers gn_numbers;
        gnumbers_node *gn_next;
};
```

In this description, the Boolean indicates whether there is more data following it. If the Boolean is FALSE, then it is the last data field of the structure. If it is TRUE, then it is followed by a `gnumbers` structure and (recursively) by a `gnumbers_list`. Note that the C declaration has no Boolean explicitly declared in it (though the `gn_next` field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.

Hints for writing the XDR routines for a `gnumbers_list` follow from the previous XDR description. Note how the primitive `xdr_pointer` is used to implement the XDR union above:

```
bool_t
xdr_gnumbers_node (xdrs, gn)
        XDR *xdrs;
        gnumbers_node *gn;
{
        return (xdr_gnumbers (xdrs, &gn->gn_numbers)  &&
            xdr_gnumbers_list (xdrs, &gp->gn_next));

bool_t
xdr_gnumbers_list (xdrs, gnp)
        XDR *xdrs;
        gnumbers_list *gnp;
{
        return (xdr_pointer (xdrs, gnp,
            SizeOf(struct gnumbers_node),
            xdr_gnumbers_node));
```

With these routines, the C stack grows linearly with respect to the number of nodes in the list. This is due to the recursion. The following routine collapses the above two mutually recursive programs into a single, non-recursive one.

```
bool_t
xdr_gnumbers_list (xdrs, gnp)
        XDR *xdrs;
        gnumbers_list *gnp;
{
        bool_t more_data;
        gnumbers_list *nextp;

        for (;;) {
             more_data = (*gnp != NULL);
             if (!xdr_bool (xdrs, &more_data)) {
                 return (FALSE) ;
             }
             if (!more_data) {
                 break;
             }
             if (xdrs->x_op == XDR_FREE) {
                 nextp = &(*gnp)->gn_next;
             }
             if (!xdr_reference (xdrs, gnp,
                     sizeof (struct gnumbers_node), xdr_gnumbers)) {
```

```
            return (FALSE);
            }
            gnp = xdrs->x_op == XDR_FREE) ?
                nextp : &(*gnp)->gn_next;
        }
        *gnp = NULL;
        return (TRUE)
}
```

The routine's first task is to find out whether there is more data or not, so that this Boolean information can be serialized. Notice that this statement is unnecessary in the XDR_DECODE case, since the value of `more_data` is not known until it is deserialized in the next statement.

The next statement XDRs the `more_data` field of the XDR union. Then if there is no more data, the routine sets this last pointer to `NULL` to indicate the end of the list, and returns `TRUE` because it is done. Note that setting the pointer to `NULL` is only important in the XDR_ENCODE case, because the pointer is already null in the XDR_ENCODE and `XDR_FREE` cases.

Next, if the direction is `XDR_FREE`, the routine sets the value of `nextp` to indicate the location of the next pointer in the list. This routine performs this task because it needs to dereference *gnp* to find the location of the next item in the list, and after the next statement the storage pointed to by *gnp* will be freed up and no longer valid. This can't be done for all directions, though, because in the XDR_DECODE direction the value of *gnp* won't be set until the next statement.

Next, the routine XDR's the data in the node using the primitive `xdr_reference`. `xdr_reference` is like `xdr_pointer`, which was used before, but it does not send over the Boolean indicating whether there is more data. The routine uses it instead of `xdr_pointer` because it has already XDRd this information. Notice that the XDR routine passed is not the same type as an element in the list. The routine passed is `xdr_gnumbers`, for XDRing `gnumbers`, but each element in the list is actually of the `gnumbers_node` type. The routine dosen't pass `xdr_gnumbers_gnode` because it is recursive, and instead uses `xdr_gnumbers`, which XDRs all of the non-recursive part.

Note: This method will work only if the `gn_numbers` field is the first item in each element, so that their addresses are identical when passed to `xdr_reference`.

Finally, the routine updates *gnp* to point to the next item in the list. If the direction is `XDR_FREE`, the routine sets it to the previously saved value; otherwise, the routine can dereference *gnp* to get the proper value. This non-recursive routine is far less likely to overflow the C stack. It also runs more efficiently since a lot of procedure call overhead has been removed. Most lists are small, however (in the hundreds of items or less), and the recursive version should be sufficient for them.

Related Information

List of XDR Examples on page 3–28.

Understanding the XDR Subroutine Format on page 3–3, Using the XDR Library on page 3–4, Understanding the XDR Language Specification on page 3–5, Understanding XDR Data Types on page 3–8, Understanding XDR Library Filter Primitives on page 3–17, Understanding XDR Non–Filter Primitives on page 3–20.

Example Showing the Justification for Using XDR

Consider two programs, **writer** and **reader**. The **writer** program is written as follows:

```
#include <stdio.h>
main()          /*  writer.c  */
{
   long i;
   for (i = 0; i < 8; i++) {
       if (fwrite((char *)&i, sizeof(i), 1, stdout) != 1) {
          fprintf(stderr, "failed!\n");
          exit(1);
       }
   }
   exit(0);
}
```

The **reader** program is written as follows:

```
#include <stdio.h>
main()          /* reader.c  */
{
   long i, j;
   for (j = 0; j < 8; j++) {
      if (fread((char *)&i, sizeof (i), 1, stdin) != 1) {
         fprintf(stderr, "failed!\n");
         exit(1);
      }
      printf("%ld ", i);
   }
   printf("\n");
  exit(0);
}
```

The two programs appear to be portable, because (a) they pass **lint** checking, and (b) they exhibit the same behavior when executed on two different hardware architectures, an IBM and a VAX.

Piping the output of the **writer** program to the **reader** program gives identical results on an IBM or a VAX as follows:

```
ibm%   writer | reader
0 1 2 3 4 5 6 7
ibm%
vax%   writer | reader
0 1 2 3 4 5 6 7
vax%
```

The following are the results if the first produces data on an IBM, and the second consumes data on a VAX:

```
ibm%   writer | rsh vax reader
0 16777216 33554432 50331648 67108864 83886080 100663296
117440512
ibm%
```

Identical results can be obtained by executing **writer** on the VAX and **reader** on the IBM. These results occur because the byte ordering of long integers differs between the VAX and the IBM, even though word size is the same. Note that 16777216 is 2^{24} — when four bytes are reversed, the 1 winds up in the 24th bit.

Whenever data is shared by two or more machine types, there is a need for portable data. Programs can be made data-portable by replacing the **read** and **write** system calls with calls to an XDR library routine **xdr_long**, a filter that knows the standard representation of a long integer in its external form.

The following are the revised versions of the **writer** program:

```
#include <stdio.h>
#include <rpc/rpc.h>      /*  xdr is a sub-library of rpc   */
main()          /*  writer.c   */
{
   XDR xdrs;
   long i;
   xdrstdio_create(&xdrs, stdout, XDR_ENCODE);
   for (i = 0; i < 8; i++) {
      if (!xdr_long(&xdrs, &i)) {
         fprintf(stderr, "failed!\n");
         exit(1);
      }
   }
   exit(0);
}
```

The following are the results of the **reader** program:

```
#include <stdio.h>
#include <rpc/rpc.h>    /*  xdr is a sub-library of rpc   */
main()                  /*  reader.c   */
{
   XDR xdrs;
   long i, j;
   xdrstdio_create(&xdrs, stdin, XDR_DECODE);
   for (j = 0; j < 8; j++) {
      if (!xdr_long(&xdrs, &i)) {
        fprintf(stderr, "failed!\n");
        exit(1);
      }
      printf("%ld ", i);
      }
   printf("\n");
   exit(0);
}
```

The new programs were executed on an IBM, on a VAX, and from an IBM to a VAX. The results are shown below.

```
ibm%   writer | reader
0 1 2 3 4 5 6 7
ibm%
vax%   writer | reader
0 1 2 3 4 5 6 7
vax%
ibm%   writer | rsh vax reader
0 1 2 3 4 5 6 7
ibm%
```

In addition to integers, arbitrary data structures require portability considerations, particularly with respect to alignment and pointers. Alignment on word boundaries may cause the size of a structure to vary from machine to machine. Pointers, which are convenient to use, have no meaning outside the machine where they are defined.

Related Information

List of XDR Examples on page 3–28.

Using the XDR Library on page 3–4.

Example Showing the Use of Pointers in XDR

If a structure containing a person's name and a pointer to a `gnumbers` structure which in turn has the person's gross assets and liabilities, the structure can be written as follows:

```
struct pgn {
    char *name;
    struct gnumbers *gnp;
};
```

The corresponding XDR routine for this structure is:

```
bool_t
xdr_pgn(xdrs, pp)
    XDR *xdrs;
    struct pgn *pp;
{
    if (xdr_string(xdrs, &pp->name, NLEN) &&
      xdr_reference(xdrs, &pp->gnp,
      sizeof(struct gnumbers), xdr_gnumbers))
        return(TRUE);
    return(FALSE);
}
```

Related Information

List of XDR Examples on page 3–28.

Understanding XDR Library Filter Primitives on page 3–17.

Example Using an XDR

In the following example, a person's gross assets and liabilities are to be exchanged among processes. These values are important enough to warrant their own data type:

```
struct gnumbers {
   long g_assets;
   long g_liabilities;
};
```

The corresponding XDR routine describing this structure would be:

```
bool_t      /*  TRUE is success, FALSE is failure  */
xdr_gnumbers(xdrs, gp)
   XDR *xdrs;
   struct gnumbers *gp;
{
   if (xdr_long(xdrs, &gp->g_assets) &&
      xdr_long(xdrs, &gp->g_liabilities))
      return(TRUE);
   return(FALSE);
}
```

Note that the parameter `xdrs` is never inspected or modified; it is only passed on to the subcomponent routines. It is imperative to inspect the return value of each XDR routine call, and to give up immediately and return `FALSE` if the subroutine fails.

This example also shows that the type `bool_t` is declared as an integer whose only values are `TRUE` (1) and `FALSE` (0). This document uses the following definitions:

```
#define bool_t     int
#define TRUE       1
#define FALSE      0
```

Keeping these conventions in mind, the `xdr_gnumbers` routine can be rewritten as follows:

```
xdr_gnumbers(xdrs, gp)
   XDR *xdrs;
   struct gnumbers *gp;
{
   return(xdr_long(xdrs, &gp->g_assets) &&
      xdr_long(xdrs, &gp->g_liabilities));
}
```

Related Information

List of XDR Examples on page 3–28.

Using the XDR Library on page 3–4.

Example Using an XDR Array

The following four examples illustrate XDR arrays:

Example A

A user on a networked machine can be identified by the machine name (using the **gethostname** routine), the user's UID (using the **geteuid** routine), and the group numbers to which the user belongs (using the **getgroups** routine). A structure with this information and its associated XDR routine could be coded as follows:

```
struct netuser {
   char     *nu_machinename;
   int      nu_uid;
   u_int    nu_glen;
   int      *nu_gids;
};
#define NLEN 255     /*  machine names < 256 chars   */
#define NGRPS 20     /*  user can't be in > 20 groups   */
bool_t
xdr_netuser(xdrs, nup)
   XDR *xdrs;
   struct netuser *nup;
{
   return(xdr_string(xdrs, &nup->nu_machinename, NLEN) &&
      xdr_int(xdrs, &nup->nu_uid) &&
      xdr_array(xdrs, &nup->nu_gids, &nup->nu_glen,
          NGRPS, sizeof (int), xdr_int));
}
```

Example B

To code a routine to use fixed-length arrays, the above example can be rewritten as follows:

```
#define NLEN 255
#define NGRPS 20
struct netuser {
       char *NUMachineName;
       int nu_uid;
       int nu_gids;
};
bool_t
xdr_netuser (XDRS, nup
       XDR *xdrs;
       struct netuser *nup;
{
       int i;
       if (!xdr_string(xdrs,&nup->NUMachineName, NLEN))
        return (FALSE);
       if (!xdr_int (xdrs, &nup->nu_uid))
        return (FALSE);
       for (i = 0; i < NGRPS; i+++) {
         if (!xdr_int (xdrs, &nup->nu_uids[i]))
             return (FALSE);
       }
       return (TRUE);
}
```

Example C

A party of network users could be implemented as an array of **netuser** structure. The declaration and its associated XDR routines are as follows:

```
struct party {
    u_int p_len;
    struct netuser *p_nusers;
};
#define PLEN 500    /*  max number of users in a party  */
bool_t
xdr_party(xdrs, pp)
    XDR *xdrs;
    struct party *pp;
{
    return(xdr_array(xdrs, &pp->p_nusers, &pp->p_len, PLEN,
        sizeof (struct netuser), xdr_netuser));
}
```

Example D

The well-known parameters to `main`, `argc`, and `argv` can be combined into a structure. An array of these structures can make up a history of commands. The declarations and XDR routines may have the following syntax:

```
struct cmd {
    u_int c_argc;
    char **c_argv;
};
#define ALEN 1000   /*  args cannot be > 1000 chars  */
#define NARGC 100   /*  commands cannot have > 100 args  */
struct history {
    u_int h_len;
    struct cmd *h_cmds;
};
#define NCMDS 75    /*  history is no more than 75 commands  */
bool_t
xdr_wrap_string(xdrs, sp)
    XDR *xdrs;
    char **sp;
{
    return(xdr_string(xdrs, sp, ALEN));
}
bool_t
xdr_cmd(xdrs, cp)
    XDR *xdrs;
    struct cmd *cp;
{
    return(xdr_array(xdrs, &cp->c_argv, &cp->c_argc, NARGC,
        sizeof (char *), xdr_wrap_string));
}
bool_t
xdr_history(xdrs, hp)
    XDR *xdrs;
    struct history *hp;
{
    return(xdr_array(xdrs, &hp->h_cmds, &hp->h_len, NCMDS,
        sizeof (struct cmd), xdr_cmd));
}
```

Related Information

List of XDR Examples on page 3–28.

Understanding the XDR Subroutine Format on page 3–3, Using the XDR Library on page 3–4, Understanding the XDR Language Specification on page 3–5, Understanding XDR Data Types on page 3–8, Understanding XDR Library Filter Primitives on page 3–17.

Example Using an XDR Data Description

Here is a short XDR data description of a `file` that might be used to transfer files from one machine to another.

```
const MAXUSERNAME = 32;     /*  max length of a user name  */
const MAXFILELEN = 65535;   /*  max length of a file       */
const MAXNAMELEN = 255;     /*  max length of a file name  */
/*
 * Types of files:
 */
enum filekind {
    TEXT = 0,        /*  ascii data   */
    DATA = 1,        /*  raw data     */
    EXEC = 2         /*  executable   */
};
/*
 * File information, per kind of file:
 */
union filetype switch (filekind kind) {
    case TEXT:
        void;                   /*  no extra information  */
    case DATA:
        string creator<MAXNAMELEN>;    /*  data creator  */
    case EXEC:
        string interpretor<MAXNAMELEN>; /* program interpretor */
};
/*
 * A complete file:
 */
struct file {
    string filename<MAXNAMELEN>; /*  name of file  */
    filetype type;               /*  info about file  */
    string owner<MAXUSERNAME>;   /*  owner of file    */
    opaque data<MAXFILELEN>;     /*  file data        */
};
```

If a user named `john` wants to store his LISP program `sillyprog` that contains just the data `(quit)`, his file can be encoded as follows:

XDR Data Description Table			
Offset	**Hex Bytes**	**ASCII**	**Description**
0	00 00 00 09	...	Length of filename = 9
4	73 69 6c 6c	sill	Filename characters
8	79 70 72 6f	ypro	... and more characters ...
12	67 00 00 00	g...	... and 3 zero–bytes of fill
16	00 00 00 02	...	Filekind is EXEC = 2
20	00 00 00 04	...	Length of owner = 4
24	6c 69 73 70	lisp	Interpretor characters
28	00 00 00 04	...	Length of owner = 4
32	6a 6f 68 6e	john	Owner characters
36	00 00 00 06	...	Length of file data = 6
40	28 71 75 69	(qui	File data bytes ...
44	74 29 00 00	t)..	... and 2 zero–bytes of fill

Related Information

List of XDR Examples on page 3–28.

Understanding the XDR Language Specification on page 3–5.

Example Using an XDR Discriminated Union

If the type of a union can be `integer`, character pointer (a `string`), or a `gnumbers` structure, and the union and its current type are declared in a structure, the declaration follows:

```
enum utype { INTEGER=1, STRING=2, GNUMBERS=3 };
struct u_tag {
    enum utype utype;   /*  the union's discriminant  */
    union {
        int ival;
        char *pval;
        struct gnumbers gn;
    } uval;
};
```

The following constructs and XDR procedure (de)serialize the discriminated union:

```
struct xdr_discrim u_tag_arms[4] = {
    { INTEGER, xdr_int },
    { GNUMBERS, xdr_gnumbers }
    { STRING, xdr_wrap_string },
    { __dontcare__, NULL }
    /*  always terminate arms with a NULL xdr_proc  */
}
bool_t
xdr_u_tag(xdrs, utp)
    XDR *xdrs;
    struct u_tag *utp;
{
    return(xdr_union(xdrs, &utp->utype, &utp->uval,
        u_tag_arms, NULL));
}
```

The `xdr_gnumbers` routine was presented in the Example Showing the Justification for Using XDR on page 3-32. The **xdr_wrap_string** routine is presented in Example D of the Example Using an XDR Array on page 3-37. The default `arm` parameter to the `xdr_union` parameter is `NULL` in this example. Therefore the value of the union's discriminant may legally take on only values listed in the `u_tag_arms` array. This example also demonstrates that the elements of the arm's array do not need to be sorted.

The values of the discriminant may be sparse (though not in this example). Assign explicit integer values to each element of the discriminant's type to document the external representation of the discriminant and guarantee that different C compilers emit identical discriminant values.

Related Information

List of XDR Examples on page 3-28.

Understanding XDR Library Filter Primitives on page 3-17, Understanding XDR Data Types on page 3-8.

Chapter 4. 3270 Host Connection Program/6000 (HCON)

The AIX 3270 Host Connection Program/6000 Licensed Program (HCON) application programming interface (API) allows AIX users to develop applications programs that communicate with a host system. This chapter/section contains information on HCON data structures, the file transfer programming interface and the HCON application program interface. Also included is information on how to install the HCON API on a host system, how to incorporate automatic logon in your HCON applications, and error and status information for program troubleshooting.

HCON Overview

The 3270 Host Connection Program/6000 (HCON) is a software package that provides communications capabilities between RISC System/6000 and IBM System/370 host computer systems. HCON provides the following programming facilities:

- File transfer programming interface
- Application Programming Interface (API)
- Implicit logon/logoff features and commands
- HCON programming examples.

File Transfer Programming Interface

The File Transfer Programming Interface permits the transfer of files between RISC System/6000 and a System/370 host. The host operating system can be either VM/CMS or MVS/TSO with the corresponding version of the IBM 3270 File Transfer Program (**IND$FILE**) installed. The application program can transfer a file from a RISC System/6000 to the host (uploading) or from the host to the RISC System/6000 (downloading). The application program can transfer either text or binary data. The program interface is a library linked with an AIX user application program.

HCON API

The Applications Programming Interface (API) to HCON allows a user to write a program that can interface with the System/370 host computer system in one of three modes.

HCON API Modes

HCON API modes are as follows:

API/API Allows an AIX application program to communicate with a host application program. The API/API mode requires the user to write an AIX application and a corresponding Host Interface application. The API/API mode is used when an AIX application uses the API to communicate with a host application that also uses the API.

API/API_T Translates data between ASCII and EBCDIC formats in addition to performing the same function as API/API mode.

API/3270 Allows an AIX application program to communicate with a System/370 host by means of standard 3270 Protocol. The API/3270 mode is used when an AIX application uses the API to communicate with a host application that assumes it is dealing with a 3270 terminal (the API/3270 application uses the emulator to communicate with the host through a 3270 session).

AIX Interface for HCON API

AIX access to the HCON API is through use of a C, Pascal, or FORTRAN language interface. The AIX Interface consists of the following categories of functions:

Session Control Enables an AIX application program to open, start, and end communication with a System/370 host application.

Message Interface Enables an AIX application program to send messages to and receive messages from a System/370 host.

API File Transfer Permits an API application to perform a file transfer through the use of the **g32_fxfer** function.

Logical Terminal Interface Allows an AIX application program to access the presentation space of the associated terminal emulator.

Host Interface for HCON API

Host Interface applications must be written in System/370 Assembler language. The Host Interface consists of the following categories of functions:

Host Session Control The 3270 assembler functions that enable a host application program to open, start, and end communication with an AIX application.

Host Message Interface The 3270 assembler functions that enable a host application program to send messages to and receive messages from AIX.

HCON provides a set of header files that the application program must include when incorporating any of the HCON API functions. The header files (also called include files) contain data structures required by the AIX HCON functions.

Logon/Logoff Facilities

Before information can be exchanged between a RISC System/6000 and a host system, the user must log on to the host system with a valid host user ID and a password. HCON supports two logon methods: explicit logon and implicit logon.

Explicit logon requires the user to invoke the **e789** program and log on to the host system.

Implicit logon requires that the user provide logon information to the file transfer or API application program being invoked. HCON supplies two facilities to allow the user to include logon information in a program: AUTOLOG and the Logon Assist Feature (LAF).

AUTOLOG and LAF may be used with explicit logon (an open emulator) if the flag in the **g32_open** function is set to one (1).

HCON Programming Examples

HCON includes program examples that illustrate the use of the API, implicit logon, and file transfer programming facilities. Program examples fall into the following categories:

File Transfer Program Interface File transfer programming examples illustrate the use of the **fxfer** function.

Application Program Interface Application Program Interface (API) examples illustrate the use of the AIX and Host functions. In addition to the example fragments provided to show how each API function works, HCON also provides three example programs illustrating the use of more than one function in an application.

Logon Assist Feature Scripts Illustrate implicit logon. The **/usr/lib/hcon** directory contains the **g_log.vm** and **g_log.mvs** sample LAF scripts.

AUTOLOG profiles Provide samples of implicit AUTOLOG profiles. The **/usr/lib/hcon** directory contains the sample AUTOLOG profiles: **SYStso**, **SYSvm1**, and **SYSvm2**.

HCON Terms

Session

A session is a period of activity with an IBM System/370 host computer. This period begins as soon as the host and HCON begin to exchange greetings. This session is separate from the logon activity. The sequence of communication events between the device driver and the controller does not constitute a session. A host application is always in control of a session. This application can be the host operating system or a user-written API program. The 3270 interface is a master/slave interface, with the master being the host computer and the slave being the RISC System/6000.

Logical Path

A logical path is a bi-directional communication route between the RISC System/6000 and the IBM System/370 host computer. Over this route flow the 3270 commands as well as messages from the host to the RISC System/6000 and from the RISC System/6000 to the host.

Physical Path

A physical path refers to a specific adapter and communication cable. A physical path supports 0 or more logical paths. The configuration of the device driver, controller, and host determines the number of paths.

Presentation Space

An internal data structure that contains all information or data that appears on an e789 terminal screen. The e789 screen display reflects the information of the internal data structure.

Related Information

Understanding the File Transfer Program Interface on page 4–4, Understanding the HCON Application Programming Interfaces on page 4–8, Understanding the AIX Interface for HCON on page 4–11, Understanding the Host Interface for HCON API on page 4–15, Understanding Explicit and Implicit Logon on page 4–17, Understanding HCON Programming Examples on page 4–25.

Understanding the File Transfer Program Interface

The file transfer programming interface consists of library routines that are linked with an AIX user application program written in C, FORTRAN or Pascal. The file transfer routines are the following:

fxfer Initiates the file transfer from within an AIX application program.

cfxfer Returns the completion status of the file transfer request to the program.

The file transfer status codes are described in the table of File Transfer Error Codes.

Note: The file transfer program interface cannot be interrupted; therefore, no restart files are created. The file transfer program interface cannot invoke the restart option. If restart files exist, the file transfer program interface function returns a status code of 202.

Synchronous and Asynchronous File Transfers

Synchronous and asynchronous file transfers differ in these respects:

- Asynchronous file transfers are processed in the background. If a current file transfer is not complete, additional asynchronous file transfer requests to the same session are placed in a queue. Each request is processed after the preceding file transfer has completed.
- Synchronous file transfers are processed immediately. If a current file transfer is not completed, additional synchronous file transfer requests generate a host connection busy message.
- It is an error to attempt a synchronous transfer while any other file transfer is being processed.

Security

The File Transfer program protects the user and host logons as well as the integrity of the files. Passwords required for any host logon are solicited from the control terminal. The password is not displayed on the screen, and it is not maintained in any file. If a password is maintained in memory for any length of time, it is disguised by the **fxfer** program.

A logical path created by the file transfer program is destroyed automatically when the session is terminated by logging off, the associated queue is empty, and the time specified in the file transfer wait period within the session profile has expired. If the associated queue is not empty, the logical path is not destroyed until all the queued requests are processed and the specified wait period has expired.

The **fxfer** function does not replace existing files unless the **FXC_REPL** option is specified. This option overwrites the contents of an existing file. If the replace option is not specified and the destination file exists, the function terminates with an error condition.

The file transfer program interface includes a file protection mechanism when downloading files to the RISC System/6000, so that the destination file which exists before the file transfer begins is not destroyed or altered if the file transfer is interrupted. This protection mechanism is not available when uploading files to the System/370 host.

File Transfer Programming Header Files

All file transfer programs use a defined data structure (provided in header files) for each language. File transfer header (include) files are as follows:

/usr/include/fxfer.h	File transfer include file with structures and definitions
/usr/include/fxconst.inc	Pascal file transfer constants
/usr/include/fxfer.inc	Pascal file transfer include file with structures
/usr/include/fxhfile.inc	Pascal file transfer invocation include file.

File Transfer Data Structures

The C and Pascal Program Interface for the **fxfer** function use data structures that are defined in header files. The C Program Interface uses the **fxfer.h** file. The Pascal Interface uses the **fxfer.inc** file.

Note: The FORTRAN interface does not use include files. Check the syntax provided by the **fxfer** and **cfxfer** functions to determine how to handle the FORTRAN implementation.

fxfer.h File

The **fxfer.h** file defines the C Program Interface **fxc** structure for the **fxfer** file transfer function. The *xfer* parameter of the **fxfer** function specifies a pointer to the **fxc** structure. Each C program module that uses the **fxfer** function must include the **fxfer.h** file.

fxc Structure for C

The C Program Interface structure **fxc** is defined as follows:

```
struct fxc {
    char *fxc_src;              /* Source file name              */
    int srclength;              /* Put here for Pascal stringpt  */
    char *fxc_dst;              /* Destination file name         */
    int dstlength;              /* Put here for Pascal stringptr */
    struct fxcf {
       int   f_flags;           /* option flags                  */

        #define FXC_UP          0x0001
        #define FXC_DOWN        0x0002
        #define FXC_TNL         0x0004
        #define FXC_TCRLF       0x0008
        #define FXC_REPL        0x0010
        #define FXC_APPND       0x0020
        #define FXC_QUEUE       0x0040
        #define FXC_FIXED       0x0080

        #define  FXC_VAR         0x0100
        #define  FXC_UNDEF       0x0200
        #define  FXC_TSO         0x0400
        #define  FXC_CMS         0x0800
          char *f_logonid;       /* Logon ID                     */
            int   loglength;       /* Put here for Pascal stringptr */
            int   f_lrecl;         /* Logical record length        */
            int   f_blksize;       /* Block size                   */
```

```
            struct fxcs {
                int s_space;         /* Allocation space           */
                int s_increment;     /* Allocation space increment */
                int s_unit;          /* Unit of allocation         */
            #define FXC_TRACKS    -1    /* Tracks                  */
            #define FXC_CYLINDERS -2    /* Cylinder                */
                 } f_s;
              fxc_opts;
            };
struct fxs {
   int  fxs_bytcnt;           /* Byte count                     */
   char *fxs_src;             /* Source file name               */
   int  srclen;               /* Put here for Pascal stringptr  */
   char *fxs_dst;             /* Destination file name          */
   int  dstlen;               /* Put here for Pascal stringptr  */
   char *fxs_ctime;           /* Destination file creation time */
   int  timelen;              /* Put here for Pascal stringptr  */
   int  fxs_stat;             /* Status code                    */
   int  fxs_errno;            /* Errno                          */
};

struct fxp {
   char *prof_id;             /* Profile id                     */
   int  proflen;              /* Put here for Pascal stringptr  */
};
```

fxfer.inc Header File

The **fxfer.inc** file defines the **fxc** record format for the Pascal Program Interface and is used by the **fxfer** file transfer function. Each Pascal program module that uses the **fxfer** function must include the **fxfer.inc** file, the **fxconst.inc** file, and the **fxhfile.inc** file.

fxc Declarations for Pascal

The **fxconst.inc** header file includes the external declarations for the file transfer Pascal interface routines: **pfxfer** and **pcfxfer**. The **fxhfile.inc** is the Pascal file transfer invocation file for **pfxfer** and **pcfxfer**. The **fxfer.inc** file contains the **fxs** and **fxc** declarations for the Pascal interface routines.

The **fx_statxxxxxx** status file contains the status of each file transfer request made by the application program. This information is placed in the $HOME directory.

C and Pascal Options

The options for the File Transfer Program Interface C structures and Pascal record declarations are as follows:

f_logonid The **f_logonid** option is a string consisting of the host logon ID. As an option, the host logon ID may be followed by a comma and a list of three (3) variables, separated by commas. This list is passed to the implicit logon procedure.

At run time, the operator will be asked to enter the password. The host logon session is maintained for subsequent file transfers, thus eliminating the need to logon again. The length of time the logon session is maintained is determined by the file transfer wait period in the HCON session profile variable.

FXC_APPND Appends the file specified by the source file to the destination file, if the destination file exists when the **FXC_APPND** flag is set in **fxc_opts.f_flags**. This option is ignored if the destination file does not exist.

FXC_CMS Specifies the host as VM/CMS when the **FXC_CMS** flag is set in **fxc_opts.f_flags**. The user must specify the correct host operating system. The file transfer program does not distinguish between the two host operating systems.

FXC_DOWN Downloads the file from a host file to an AIX file when this is set in **fxc_opts.f_flags**.

FXC_QUEUE Executes the file transfer asynchronously as a background process when this is set in **fxc_opts.f_flags**. If any file transfers have not completed, the current transfer request is queued. If this option is not specified, the file transfer operation is synchronous.

FXC_REPL Replaces an existing file on the host (upload) or replaces an existing AIX file (download), when the **FXC_REPL** flag is set in **fxc_opts.f_flags**.

FXC_TSO Specifies the host as MVS/TSO when this flag is set in **fxc_opts.f_flags**. The user must specify the correct host operating system. The file transfer program does not distinguish between the two host operating systems.

FXC_TNL Translates EBCDIC to ASCII when downloading files if this is set in **fxc_opts.f_flags**. During uploading, it translates ASCII to EBCDIC. This option assumes the file is a text file. The default is no translation. The AIX new-line character is the line delimiter. This option is used when transferring AIX formatted text files.

FXC_TCRLF Performs the same function as the **FXC_TNL** flag when this is set in **fxc_opts.f_flags** except that the line delimiter is the character sequence CR-LF. This option is used to translate PC-DOS files. A PC-DOS end-of-file character is inserted at the end of the downloaded file.

FXC_UP Uploads the file from the AIX file to the host file when this is set in **fxc_opts.f_flags**.

Host File Flags

The following flags specify host file characteristics. They can only be used to upload files.

f_blksize Specifies the non-zero block size of the host data set. This option can only be used in the MVS/TSO environment. For new files, the default is the logical record length. This flag is ignored if the file is being appended.

f_lrecl Specifies the non-zero logical record length of the host file. For new files, the default is 80. For variable-length records, the **f_lrecl** flag is the maximum size of the record. This flag is ignored if the file is being appended.

FXC_FIXED Specifies fixed-length records when set in **fxc_opts.f_flags**. This is the default if neither the **FXC_VAR**, the **FXC_TNL**, nor the **FXC_TCRLF** flag is set. This flag is ignored if the file is being appended.

FXC_UNDEF Specifies records of undefined length when set in **fxc_opts.f_flags**. This option can only be used in the MVS/TSO environment. This flag is ignored if the file is being appended.

FXC_VAR Specifies variable-length records when set in **fxc_opts.f_flags**. This is the default if the **FXC_FIXED** flag is not set and either the **FXC_TNL** or the **FXC_TCRLF** flag is set. This flag is ignored if the file is being appended.

s_space Specifies the non-zero number of units of space to be allocated for a new data set. This option can only be used in the MVS/TSO environment. The **s_space** field has the following optional sub–fields:

s_increment Specifies the number of units of space to be added to the data set each time the previously allocated space is filled.

s_unit Specifies the unit of space. A value of **FXC_TRACKS** indicates the unit of allocation is tracks. A value of **FXC_CYLINDERS** indicates the unit of allocation is cylinders. Otherwise, the **s_space** field specifies the average block size (in bytes) of the records that are to be written to the data set. If the **s_space** field is zero, the default unit of allocation is the value specified by the **f_blksize** field. If the **f_blksize** field is not specified, the the host file transfer program uses the default value of 80.

Related Information

The **fxfer** function, **cfxfer** function, and **fxfer** command.

Understanding Explicit and Implicit Logon on page 4–17, Understanding the HCON API on page 4–8.

Understanding the HCON File Transfer Process in *Communication Concepts and Procedures*.

Understanding the HCON Application Program Interface (API)

The HCON Application Programming Interface (API) provides high-level access to the communication link for program-to-program communication between AIX and the host application. The API consists of the following elements:

- An object library on AIX
- A MACLIB, a TXTLIB, and two I/O modules (VM only) on the host System/370 computer.

Note: The API may be used for communication between AIX and either the VM/CMS or MVS/TSO environments.

Both the AIX application program and the host application program must use the HCON API commands to communicate with one another.

The API program provides these functions:

- Program-to-program communication. AIX applications may communicate with host applications in a VM/CMS or MVS/TSO environment.

- Write and read functions for transmission of messages between the AIX application and the System/370 host applications. Messages are simple byte strings containing arbitrary data and may be up to the length specified by the maximum I/O buffer size in the session profile.
- Optional data translation between EBCDIC (in the host) and ASCII (in AIX) in messages.
- Automatic logon of the AIX application to the host and initiation of the host application.
- Session control functions to start and stop sessions.
- An interface to the terminal emulator, so that an AIX application can invoke a host application that assumes it is communicating with a 3278/79 terminal.

The following rules apply to application programs using the API:

- Host application names are in uppercase letters when they are used to describe functions. AIX application names must be specified in lowercase letters.
- An application may use the API to establish a session. The session is unique to the AIX-application/Host-application pair that established it.
- The HCON API library reserves the prefix **g32** for API functions. Do not use this prefix for naming user application programs.
- By definition, control of a session cannot be passed to another process. A logical path however, can be transferred from one AIX process to another.

Sample Flows of API Programs

The HCON API allows an AIX application program to communicate with an application program located at the System/370 host (VM/CMS or MVS/TSO), using HCON API functions. Applications can be written in both environments and communicate together over a 3270 session. The applications must be synchronized; when an AIX application issues a read, the host must issue a write, and vice versa.

The following is a sample program flow for an AIX application program and a Host Interface program using the API/API or API/API_T mode. It also matches API functions with Host Interface functions, showing how HCON applications are synchronized.

```
AIX Application                  S/370 Host Application
  g32_openx                            ────────
  g32_alloc                            G32ALLOC
  g32_read                             G32WRITE
  g32_write                            G32READ
  g32_get_status                       ────────
  g32_dealloc                          G32DLLOC
  g32_close                            ────────
```

HCON includes two AIX and host programs that use the session control and message interface functions.

The following is a sample program flow for an AIX application program using the API/3270 mode:

```
AIX Application
  g32_openx
  g32_alloc
  g32_notify
  g32_search
  g32_send_keys
  g32_get_data
  g32_get_cursor
  g32_notify
  g32_dealloc
  g32_close
```

Note: API/3270 mode applications should not invoke Host Interface applications.

Related Information

Understanding Explicit and Implicit Logon on page 4–17, Understanding the File Transfer Program Interface on page 4–4, Understanding HCON Programming Examples on page 4–25.

HCON Overview for System Management on page 4–1.

Understanding the AIX Interface for HCON API

The HCON Application Programming Interface (API) includes a set of AIX library subroutines that are linked with AIX applications that use it. The subroutines are C function calls that interface to API and file transfer functions. The **/usr/lib/libg3270.a** library contains these subroutines. The **/usr/include/g32_api.h** file contains associated symbol definitions and structures. Any C program using API functions should include the **g32_api.h** file. The API also supports Pascal and FORTRAN. The **/usr/lib/libg3270p.a** and **/usr/lib/libg3270f.a** files contain the Pascal and FORTRAN equivalents to the C library.

The components of the HCON AIX Interface are as follows:

- Header files (include files)
- Session control
- Message interface
- File transfer
- Logical terminal interface.

AIX Header Files

The C and Pascal Program Interface for the AIX API use data structures that are defined in header files. The FORTRAN Program interface contains data structure equivalents which are also declared in header files. The AIX header files are as follows:

/usr/include/g32const.inc	Pascal API constants include file
/usr/include/g32hfile.inc	Pascal API include file with external definitions
/usr/include/g32keys.inc	Pascal API include file for API/3270 mode
/usr/include/g32types.inc	Pascal API data types include file
/usr/include/g32_api.h	API include file
/usr/include/g32_keys.h	API keys include file.

AIX Data Structures

All AIX API programs use a defined data structure **g32_api** (provided in header files) for each language. An important field in this structure is the **eventf** field. This field is the file descriptor of the communication device special file if the application opens a session in API/API or API/API_T mode.

If the application opens a session in API/3270 mode, the **eventf** field is a message queue ID. The application may use this and the **g32_notify** function to instruct the emulator to notify the application when it receives data from the host.

C Language Structures

Each module that uses the API must include the **/usr/include/g32_api.h** file. This file contains the **g32_api** data structure. The fields in the **g32_api** structure are:

lpid Logical path id

errcode Error code indicator

xerrinfo Extra error information

row Row number

column Column number

length Length for pattern

eventf Message queue ID/file descriptor

maxbuf Maximum buffer size

timeout Timeout of host response.

Pascal Language Structures

Each module that uses the Pascal Program Interface to the API must include the **/usr/include/g32types.inc**, **/usr/include/g32hfile.inc**, and **/usr/include/g32const.inc** files. The **/usr/include/g32types.inc** file contains the Pascal equivalent to the **g32_api** structure. The following example illustrates the use of the Pascal header files:

```
program example(input, output);
    const
        %include /usr/include/g32const.inc
        /* user's constant definitions */
    type
        %include /usr/include/g32types.inc
        /* user's type definitions */
    var
        User_Buffer : packed array[1..100] of char;
        API_BUF_PTR : integer;
        /* user's variable declarations */
    %include /usr/include/g32hfile.inc
    /* user's external function declarations */
    begin
        API_BUF_PTR = addr(User_Buffer);
        /* user's program */
    end
```

The API_BUF_PTR declaration must be an integer and must be assigned the address of the User_Buffer declaration.

The **g32hfile.inc** file contains all the external declarations for each of the API Pascal interface routines. The two declarations User_Buffer and API_BUF_PTR must be declared by the programmer. The User_Buffer declaration can be called anything, but it must be a packed array. The API_BUF_PTR declaration must point to the User_Buffer declaration. This ensures that the API interface and the programs have the same types.

FORTRAN Language Structures

The following declaration must be used in FORTRAN programs. It is equivalent to the C **g32_api** structure.

```
INTEGER AS(9)
```

An array can have any name, but the size of the array must be nine. Constants are recommended for different subscripts. For example:

```
INTEGER, ERRCD, XERR, ROW, COL, LEN, EVENT,
DATA, MAXB, TIMEOUT, LPID, ERRCD, XERR,
ROW COL, LEN, /1, 2, 3, 4, 5, 6, 7, 8, 9/
```

The following statements use constants to print the value of the timeout.

```
        WRITE (*,1000) AS(TIMEOUT)

1000    FORMAT ('The TIMEOUT is ->' I)
```

Reserved Variables

The following is a list of reserved variables that the API uses internally. An AIX application should not attempt access to these variables.

- Any variable beginning with the prefix **g32_**
- Any variables beginning with the prefix **API_**
- **sessions**
- **in_qid**
- **out_qid**
- **q_ctoa**
- **q_atoe.**

HCON API Session Control

The AIX Interface to the HCON uses the following session control functions:

g32_open Attaches to a session. If session does not exist, an attempt is made to start the session (i.e. implicit). The user is logged on to the host if requested.

g32_openx Attaches to a session. If session does not exist, an attempt is made to start the session (i.e. implicit).

g32_alloc Initiate interaction with a host application.

g32_dealloc End interaction with a host application.

g32_close Detach from session and terminate session if created by the **g32_open** function.

HCON API Message Interface

The message interface functions are used to exchange messages with the host application. They may be used only if the session is in API/API mode or API/API_T mode.

Message interface functions are as follows:

g32_write Sends a message to a host application.

g32_read Receives a message from a host application.

g32_get_status Returns status information on I/O operation.

HCON API File Transfer

The API file transfer function allows a file transfer to occur to the active session while running an API application.

g32_fxfer Issues a file transfer to the host. **g32_open** must have been executed and all host applications terminated through the **dealloc** function for the **g32_fxfer** to run.

HCON API Logical Terminal Interface (LTI)

Each terminal emulator has a presentation space that contains the image of the data displayed on the terminal screen. This same presentation space is associated with and available to an application using the API.

The logical terminal interface functions are used for access to the presentation space of the terminal emulator and for sending keystrokes to the emulator as if the keystrokes were coming directly from the keyboard. The logical terminal interface may be used only if the API is operating in API/3270 mode.

The AIX application should wait for the host display to complete its redraw before issuing a logical terminal interface function to synchronize input and output on the host.

The logical terminal interface does limited error checking of parameters. The AIX application should verify function parameters before issuing these commands.

Logical terminal functions are as follows:

g32_search Searches for a character pattern in the session presentation space.

g32_get_cursor Returns the current cursor position in the session presentation space.

g32_get_data Obtains current display data from the session presentation space.

g32_send_keys Sends one or more keystrokes to the terminal emulator.

g32_notify Turns data notification on or off.

AIX Interface Errors

AIX Interface errors are explained in the Summary of AIX API Errors table.

Related Information

The AIX session control subroutines are the **g32_open** function, **g32_openx** function, **g32_alloc** function, **g32_dealloc** function, and **g32_close** function.

The message interface functions are the **g32_write** function, **g32_read** function, and **g32_get_status** function.

The file transfer function is the **g32_fxfer** function.

The logical terminal functions are the **g32_search** subroutine, **g32_get_cursor** subroutine, **g32_get_data** subroutine, **g32_send_keys** subroutine, and **g32_notify** subroutine.

Understanding Explicit and Implicit Logon on page 4–17, Understanding the File Transfer Program Interface on page 4–4, Understanding HCON Emulator Sessions in *Communication Concepts and Procedures*, and Pascal Language Information in *XL Pascal Language Reference* .

Understanding the Host Interface for HCON API

The host interface is a set of library routines that are linked with user-supplied 370 Assembler applications. Each routine implements an HCON API function. A macro library is provided for Assembler programmers to call these HCON API functions. HCON includes versions of the host interface for VM/CMS and MVS/TSO environments.

The HCON API must be installed on the host System/370. Two procedures describe how to install HCON on the host. The procedures are:

- How to Install the HCON MVS/TSO Host API
- How to Install the HCON VM/CMS Host API.

The How to Compile a Host API Program procedure on page 4–69 describes how to compile an application containing Host Interface functions.

Host interface functions are grouped into session control and message interface functions. These functions are analogous to the **g32_alloc**, **g32_dealloc**, **g32_read**, and **g32_write** AIX Interface functions.

The HCON host interface is made up of the following:

- Host session control
- Host message interface.

Host Session Control

The host session control functions are as follows:

G32ALLOC Initiates interaction with an AIX application.

G32DLLOC Ends interaction with an AIX application.

Host Message Interface

Message interface functions are used to exchange data with the AIX application. They may be used only while in API/API mode or API/API_T mode. Message interface functions for the host interface are:

G32READ Receives a message from an AIX application.

G32WRITE Sends a message to an AIX application.

Host Interface Errors

A DSECT is provided in the macro library to define the error values referred to by the command. The DSECT is brought in by coding **G32DATA DSECT=YES** in the host application. The error codes are defined as negative numbers. A return code that is equal to or greater than 0 (zero) indicates a normal or successful completion.

Host Interface errors are explained in the Summary of Host API Errors tables.

Related Information

Host interface functions are the **G32ALLOC**, **G32DLLOC** function, **G32READ** function, and **G32WRITE** function.

Understanding the HCON API on page 4–8.

Summary of Host API Errors.

Understanding Explicit and Implicit Logon

Before information can be exchanged between a RISC System/6000 and a host system, the user must log on to the host system with a valid host user ID and a password. There are two ways to log on to a host system: explicit logon and implicit logon.

Note: The HCON API allows the user to use AUTOLOG explicitly by setting the **g32_open** (or **g32_openx**) flag parameter to one (1). Explicit AUTOLOG means an emulator is up, but the host is not logged onto when the application is executed.

Explicit Logon

Performing an explicit logon requires the following steps:

1. Enter `e789` session name from an AIX shell.

   ```
   e789a
   ```

2. After the 3270 emulator screen is displayed, log on to the host system.
3. Initiate an AIX subshell, after logging on to the host operating system, by pressing the emulator SHELL key (default is Ctrl–C).

   Note: From the subshell, the user can execute a file transfer/API application. The user may also execute a file transfer/API application explicitly from another shell as long as the session profile is specified or the *SNAME* environment variable is set to the session name.

An explicit logoff requires the following steps:

1. Exit the emulator subshell, once the communications process has completed.
2. Log off the host system.
3. Terminate the emulator session, by pressing the emulator QUIT key, (the default is Ctrl-D pressed twice).

Implicit Logon

For implicit logon, the user must provide logon information to the communications application being invoked. Before an implicit logon, the user must develop either a Logon Assist Feature (LAF) script or an AUTOLOG profile. Following are the steps for an implicit host logon:

1. Invoke the desired communications process from an AIX shell by entering the **fxfer** file transfer command from the command line or by entering the name of a file transfer or API application.
2. The user is prompted for a host user ID and any other host logon parameters if they are not specified within the session profile or in the **fxfer**/API parameter lists.
3. The user is prompted for a host password.

The communications process logs on to the host computer, transfers the data, and logs off the host computer.

The implicit logon procedures are subroutines linked into the **fxfer** command, **fxfer** applications, and API applications. The implicit logon procedures contain two methods of operations.

- AUTOLOG is a non-programming, menu-driven interface. The **genprof** command generates the input file for the AUTOLOG procedure in a file, and the AUTOLOG interface reads the file when an HCON API program or a file transfer operation attempts to log on and log off the host computer.
- The Logon Assist Feature (LAF) is a high-level language that enables a user to write a program that logs on and off of a host computer.

Both the AUTOLOG and LAF procedures use the API logical terminal interface functions for access to the presentation space of the terminal emulator and for sending keystrokes to the host. Utility commands aid the user in creating, testing, or linking the implicit logon procedures.

AUTOLOG procedure

The AUTOLOG procedure allows the user to create a logon procedure with a menu-driven utility. This method is intended for users with no programming experience. The utility is easy to use and does not require writing a program.

If the AUTOLOG procedure is used, the **autolog.o** object module must be linked to the user's API application program. The user must then provide an AUTOLOG profile, which contains all the information necessary to perform logon and logoff operations. Use the **genprof** command to create the profile. HCON also provides sample AUTOLOG profiles that contain the appropriate information for the AUTOLOG procedure.

Note: The HCON installation process links the **fxfer** command with the **autolog.o** object module. Therefore, a file transfer program using AUTOLOG does not need to perform a link to the **autolog.o** object module. It also links **tlaf** with AUTOLOG, so any AUTOLOG profile can be tested.

LAF method

The Logon Assist Feature (LAF) method provides a language to develop LAF logon and logoff scripts. LAF statements help use and configure system response. The user must develop LAF scripts that contain all the necessary information to perform logon and logoff operations. Once the user develops a LAF script, the script must be converted to C, compiled, and linked to the API application program. The process is as follows:

1. The **genlaf** command converts the LAF scripts to the C language **laf.c** source file.
2. Compiling the **laf.c** source file produces a **laf.o** object file. To compile **laf.c**, enter the following:

   ```
   cc -c laf.c
   ```
3. The **laf.o** object file is linked to the API application program.

Note: You can also use the **apilaf** command which performs all three steps.

Explicit and Implicit File Transfer Differences

Explicit and implicit file transfers differ in these respects:

- In the explicit transfer, no input from the user is needed other than the file transfer option parameters.
- In the implicit transfer, a logon script program suitable to the user's logon procedure is required. The user is prompted for the logon ID string, which should contain the logon ID for LAF scripts or the logon ID, the script name, and the optional AUTOLOG parameters for AUTOLOG. The user is not prompted for the logon ID string if the host logon ID is specified for LAF or the host logon ID, AUTOLOG script name, and optional AUTOLOG parameters are specified for AUTOLOG within the session profile or with the **–x** option.
- Recovery is possible only with implicit transfers. The **dfxfer** file transfer process uses the logon script, the user logon ID, and the password to log back on to the host and execute the file transfer. The file transfer occurs only if the link is recovered within the time specified by the file transfer recovery time in the session profile.
- Explicit transfers restart both implicit and explicit restart files. The **x_fxfer.r** explicit restart file requests are transferred to the currently logged-on session, and the **i_fxfer.r** implicit restart file requests are transferred using the session specified by the **–n** option and the appropriate logon ID is saved in the restart file. The user is prompted for the password.
- Implicit transfer can only restart the **i_fxfer.r** implicit restart file because the logon IDs of the explicit transfer requests are not known.
- By default, the **dfxfer** process is linked with the AUTOLOG implicit logon procedure.

Related Information

Understanding the HCON File Transfer Process in *Communication Concepts and Procedures*, Understanding the HCON API on page 4–8, Understanding the Automatic Logon Commands on page 4–25, Understanding the Logon Assist Feature on page 4–20, Understanding AUTOLOG on page 4–19.

HCON Overview for System Management on page 4–1.

Understanding AUTOLOG

If the host logon ID is set in the session profile, the user is only prompted for the password and the rest of the parameters are retrieved from the profile. If the host logon ID is not set, the user is prompted for both the host logon string and the password. The first value in the logon ID string must be the host Login ID. The second value is the AUTOLOG profile name (Mode ID). Two more optional values can be passed to the AUTOLOG procedures: **trace** and **time**.The **trace** procedure controls output to help users test the AUTOLOG profile. The **time** option allows the user to change the default time of three seconds to wait for a particular pattern to be received from the host.

If the user leaves off certain parameters from the host logon string, and they are set in the profile, they are still retrieved For example, if the user sets the **nodeid**, **trace**, and **time** parameters in the session profile, the user need only enter the host logon ID when prompted. However, a user response to a prompt always overrides a profile parameter. For example, if the user sets AUTOLOG time parameter in the session profile, but enters a different value on the prompt line, the latter value is used.

The **autolog.o** contains two procedures, **g32_logon** and **g32_logoff**, which are called by the File Transfer Program or the Application Program Interface (API).

The **g32_logon** procedure reads the following information from the AUTOLOG profile about each logon or logoff event: a pattern to be searched in the presentation space, data and keystrokes to be sent to the emulator, and the next-events list. The first event from the next-events list refers to the expected event. The expected event is one that will be invoked next upon successful completion of the last event. All others are unexpected events.

Both **g32_logon** and **g32_logoff** procedures check the presentation space for a particular pattern. If the search for the pattern is successful, the input string and keystroke are sent to the emulator. The process proceeds with the first event from the next-events list. If the search fails after a specified period of time, the process proceeds to the next unexpected event. A null event signifies the end of the logon or logoff procedure. A –1 causes a return to the previous event. If the logon or logoff process fails, the procedure terminates with an error code, which is the number of the last event processed.

The user must ensure that the host session is inactive following the **g32_logoff** procedure. That is, the logoff procedure should be terminated at the same screen normally used for logging on to the host.

Note: To ensure inactivity, a time-delayed search of the presentation space can be included in either the AUTOLOG profile or the LAF script. After the logoff and ENTER keys have been sent to the host for execution, the search routine looks for a pattern that is unique to the logon screen. Only when this pattern is found is it acceptable to exit the AUTOLOG profile or LAF script.

If the **g32_logoff** procedure is not ended properly, the next call to the **g32_logon** procedure may fail because the host screen is the last screen manipulated by the previous **g32_logoff** procedure.

Related Information

The **apilaf** command, **fxlaf** command, **genlaf** command, **genprof** command, **mtlaf** command, **tlaf** command.

Understanding Explicit and Implicit Logon on page 4–17, Understanding Automatic Logon Commands on page 4–25, Understanding the File Transfer Program Interface on page 4–4, Understanding the HCON File Transfer Process in *Communication Concepts and Procedures*.

Understanding the Logon Assist Feature (LAF)

The Logon Assist Feature (LAF) helps users develop scripts for programmatic (implicit) logging on to and logging off from the host. A script:

- Sends data to the host.
- Waits for data to be received from the host.
- Looks for patterns in a presentation space.

The HCON File Transfer Program and API applications then use these scripts or programs to perform a logon and logoff.

LAF provides the usual control structures (if-then-else, while-do, repeat-until, and select), and allows users to customize their scripts during execution by passing up to 10 values in the logon ID string. The **genlaf** command generates a C program from the LAF script.

The LAF processor takes input in the LAF language and produces a C program. This program is compiled and linked with a user program that is linked with the API or file transfer. Two scripts are required: **g32_logon** and **g32_logoff**. Both scripts are input to the LAF processor in the same source module.

The host session must be inactive following the **g32_logoff** procedure. That is, the procedure should terminate at the same screen normally used for logging on to the host.

If the **g32_logoff** procedure is not ended properly, the next call to the **g32_logon** procedure may fail because the host screen is the last screen manipulated by the **g32_logoff** procedure.

LAF Language Description

The basic unit of the Logon Assist Feature (LAF) language is the token. Statements are sequences of tokens. The LAF language tokens are divided into the following categories:

- Reserved Words
- Strings
- Key Names
- Special Variables
- Language Conventions.

Reserved Words

Reserved words determine statement types. LAF reserved words are the following:

BREAK	**OTHERWISE**
DEBUG	**RECEIVE**
DO	**RECVAT**
ELSE	**REPEAT**
END	**SELECT**
EXIT	**SEND**
RECOVERY	**START**
FINISH	**TIMEOUT**
IF	**UNTIL**
MATCH	**WHEN**
	WHILE

Strings

Strings are sequences of any number of characters, delimited by either the single quote (') or double quote (") delimiters. Either delimiter may be used, but the delimiters must match. (For example, a string begun with a single quote must be terminated with a single quote.) If you want to use a single quote in a string, you must use a double quote as the delimeter, and if you want to use a double quote in a string, use the single quote as the delimiter. Strings must be on a single line; the newline character is not allowed in a string. The RECEIVE, RECVAT, MATCH, and MATCHAT functions allow certain special characters in strings. These special characters are discussed in the description of the **g32_search** function.

Key Names

Key names are special tokens that identify non-alphanumeric keys on a 3270 family terminal. Use the key name to send these key strokes to the emulator. The following are valid key names:

KEY NAMES		
ENTER	PF13	DEL (delete)
PA1	PF14	C_UP (cursor up)
PA2	PF15	C_DN (cursor down)
PA3	PF16	C_LT (cursor left)
PF1	PF17	C_RT (cursor right)
PF2	PF18	C_UUP (cursor up fast)
PF3	PF19	C_DDN (cursor down fast)
PF4	PF20	C_LLT (cursor left fast)
PF5	PF21	C_RRT (cursor right fast)
PF6	PF22	TAB
PF7	PF23	B_TAB (backtab)
PF8	PF24	CR (carriage return or newline)
PF9	CLEAR	RESET
PF10	DUP	E_INP (erase input)
PF11	FM (field mark)	E_EOF (erase EOF)
PF12	INS (insert)	T_REQ (test req or sys req)
	HOME	PRINT (screen print)

Special Variables

The LAF language uses the following special variables:

UID
Represents the host user ID. UID is an alias for ARC(0).

PW
Represents the password associated with UID. It is provided automatically by the file transfer and API.

ARG(0) – ARG(9)
Represents the first through tenth variables, respectively, in the logon ID string. The logon ID string is passed to each LAF script. ARG(0) must be in the logon ID string, but the remaining nine variables are optional. These ten arguments are used by both **g32_logon** and **g32_logoff** procedures.

ROW
Indicates the row on the screen where the target is found, after a successful scan of the presentation space. This variable is applicable to the **RECEIVE**, **RECVAT**, **MATCH**, and **MATCHAT** language statements.

COL
Indicates the column on the screen where the target is found after a successful scan of the presentation space. This variable is

applicable to the **RECEIVE**, **RECVAT**, **MATCH**, and **MATCHAT** language statements.

MATCH Indicates a successful search of the presentation space. This variable is applicable to the **RECEIVE**, **RECVAT**, **MATCH**, and **MATCHAT** language statements

RECOVERY Is set if the HCON file transfer program interface is trying to log on and recover from an error during an earlier file transfer. This variable may be tested in the LAF conditions.

LAF Language Conventions

LAF language conventions include the following categories:

Logical Operators

The logical operators used in building expressions are:

not (NOT or !)

and (AND or &&)

or (OR or ||)

Comparison Operators

The comparison operators are used in building expressions. These operators are:

is equal to (=)

is not equal to (! =)

is less than (<)

is greater than (>)

is less than or equal to (< =)

is greater than or equal to (> =)

Numbers

LAF defines a number as a sequence of the digits 0–9, possibly with a leading minus sign (–).

Parentheses

LAF uses parentheses to :

- Group terms in an expression
- Enclose a parameter list in a statement.

Commas

Commas are used to separate items in parameter lists.

Statement Terminators

LAF uses the semicolon (;) to terminate statements. All statements must end with a semicolon.

Comments

Comments are sequences of any number of characters bounded by /* and */ delimiter. Comments may stretch across several lines.

SCRIPT Statements

A LAF script consists of the following components:

- A **START** statement
- A sequence of other statements:

  SEND

  RECEIVE

  RECVAT

  MATCH

  MATCHAT

  DO–END

  WHILE

  REPEAT–UNTIL

  BREAK

  IF–ELSE

  SELECT

  EXIT

  WAIT

  DEBUG

  NODEBUG

- A **FINISH** statement.

Related Information

The **apilaf** command, **fxlaf** command, **genlaf** command, **genprof** command, **mtlaf** command, **tlaf** command.

Understanding Explicit and Implicit Logon on page 4–17, Understanding Automatic Logon Commands on page 4–25, Understanding the File Transfer Program Interface on page 4–4, Understanding the HCON File Transfer Process in *Communication Concepts and Procedures* .

Understanding the Automatic Logon Commands

HCON provides the following utility commands used for generating, testing, and using the implicit logon procedures:

apilaf Links an implicit logon procedure into a user's API application.

fxlaf Links the **fxfer** function with a LAF script or AUTOLOG to generate a file transfer program containing a implicit logon procedure.

genprof Generates the input file for the AUTOLOG procedure.

genlaf Generates a C program from a LAF script.

mtlaf Makes a test program for an AUTOLOG profile or a LAF script.

tlaf Output test program produced by the **mtlaf** command.

logform Example of the implicit logon input form.

Related Information

The **apilaf** command, **fxlaf** command, **genlaf** command, **genprof** command, **mtlaf** command, **tlaf** command.

Understanding HCON Programming Examples

The AIX applications are written in C language, and host applications are written in 370 Assembler language. The C and Assembler source code, as well as AIX executable code, are located in the AIX directories:

/usr/lib/hcon/vm For VM/CMS systems

/usr/lib/hcon/mvs For MVS/TSO systems.

File Transfer Programming Examples

File transfer programming examples are provided only in the documentation and are for illustrative purpose only. These program examples illustrating the use of the File Transfer Programming Interface are as follows:

cname Example C language program illustrating the file transfer process using the file transfer program interface. This program can found on page 4–53.

pname Example Pascal language program illustrating the file transfer process using the file transfer program interface. This program can found on page 4–56.

fname Example FORTRAN language program illustrating the file transfer process using the file transfer program interface. This program can found on page 4–55.

API Programming Examples

HCON provides three API programs to aid users in developing API applications:

g32_test Verifies API HCON installation.

g32_sampl Evaluates API/API performance.

g32_3270 Demonstrates API/3270 mode application.

The **instalapi** program installs and compiles API program examples. The **instalapi program** also creates the **g32asm** exec for VM and TSO. The **g32asm** exec assembles and creates executable load modules of the test programs: **g32test**, and **g32sampl**.

Running Example Programs in TSO

In TSO, the **g32asm** exec allows the AIX API programs to execute the test programs by keying in the program name. The **g32asm** exec must have executed before the API program can run any of the test programs.

If you log off and start a new session, the next time you try to run the test programs you will get an error (in TSO only).

To get them to execute again (in TSO only), you cannot key in their names as before.

You can either execute the **g32asm** exec again or you can execute one of the following commands:

```
call g32appl (g32test)

call g32appl (g32sampl)

call g32appl (g323270)
```

If you have not used the **g32asm** exec during your session and you add API programs to the **g32appl** data set, you must run **g32asm** exec or use one of the **g32appl** calls to run the test program.

AUTOLOG and LAF Program Examples

The **/usr/lib/hcon** directory contains the following AUTOLOG and LAF programming examples:

g_log.mvs Example LAF program for MVS/TSO host

g_log.vm Example LAF program for VM/CMS host

SYStso Example AUTOLOG script for MVS/TSO host

SYSvm1 Example AUTOLOG script for VM/CMS host

SYSvm2 Example AUTOLOG script for VM/CMS host.

Compiling Programs

HCON includes commands to assist in debugging, compiling, and linking the example programs and the Logon Assist Feature scripts. The following procedures describe how to debug, compile, and link applications and scripts.

How to Compile a File Transfer (**fxfer**) Program

How to Compile a Host API Program

How to Compile an AIX API Program

How to Use a Logon Assist Feature Script

Related Information

Understanding Explicit and Implicit Logon on page 4–17, Understanding the Logon Assist Feature (LAF) on page 4–20, Understanding the HCON API on page 4–8, Understanding the File Transfer Program Interface on page 4–4.

File Transfer Program Interface Error Codes

The following table lists the File Transfer Program Interface error codes, describes the errors, and provides suggestions for user responses.

File Transfer Program Interface Error Codes		Part 1 of 14
Error Code	**Error Description**	**User Response**
200	The file transfer program cannot attach to the HCON shared memory.	Check that the HCON resource manager is running.
202	A **RESTART** file exists. The file transfer is stopped. A programmatic file transfer cannot run while a **RESTART** file exists.	Remove the **RESTART** file (**i_fxfer.r** or **x_fxfer.r**) before trying to run the programmatic file transfer.
203	Cannot get information about the **RESTART** file. The **stat()** system call error number is (**fxs_errno**).	Check path name and permissions or use local problem reporting procedures.
209	Cannot get the path name of the current working directory.	Check path name and permissions or use local problem reporting procedures.
214	The file transfer program cannot communicate with the HCON resource manager.	Check that the HCON resource manager is running or use local problem reporting procedures.
215	Return code (**fxs_errno**) from the HCON resource manager is not correct.	Check that the HCON resource manager is running, check that the session is available or use local problem reporting procedures.
221	This user cannot run the **dfxfer** file transfer background process.	Check for the **dfxfer** process in **/usr/bin**, check permissions, or use local problem reporting procedures.
222	The **HOME** environment variable is not defined.	Set the **HOME** environment variable to your **HOME** directory.
225	Either the host link address is busy, the host link address does not exist, or there are some remaining messages on the message queues.	If the host link address is busy, try again later. If the host link address is not busy, restart the emulator and try the file transfer again or use local problem reporting procedures.
226	The host operating system specified is invalid.	Specify either TSO or CMS only.
253	Cannot create another **dfxfer** process at this time. The **fork** system call error number is (**fxs_errno**).	Check that there is a valid **dfxfer** process in the directories of the **PATH** environment variable or use local problem reporting procedures.
259	Cannot send an interrupt signal to kill the **dfxfer** file transfer daemon background process. The **kill** system call error number is (**fxs_errno**).	Kill the **dfxfer** file transfer daemon process by issueing a **kill –9** to the **dfxfer** process on the command line or use local problem reporting procedures.

File Transfer Program Interface Error Codes		Part 2 of 14
Error Code	**Error Description**	**User Response**
268	Cannot initialize the **dfxfer** file transfer background process. Cannot open the link address file. The **open** system call error number is (**fxs_errno**).	Check path name and permission or use local problem reporting procedures.
269	Cannot initialize the **e789x** emulator server process. The communication device specification in the session profile may not be correct.	Check that there is a valid **e789x** program in the directories of the **PATH** environment variable, check that the communication device specified in the session profile exists and is valid or use local problem reporting procedures.
271	Cannot initialize the **e789x** emulator server process.	Check that there is a valid **e789x** program in the directories of the **PATH** environment variable or use local problem reporting procedures.
273	Cannot initialize the **e789x** emulator server process. The emulator server process ended abnormally.	Check that there is a valid **e789x** program in the directories of the **PATH** environment variable, retry the file transfer, or use local problem reporting procedures.
277	Transfer Status: The file transfer request was cancelled. Diagnostics: There was an error sending the file transfer request to the host. Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check to make sure that the control unit is running or use local problem reporting procedures.
280	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: The wrong host operating system has been specified either on the command line or in the session profile. Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Retry the file transfer specifying the correct operating system.
281	Transfer Status: The file transfer completed with no errors. The host file records are segmented. Byte Count: (**fxs_bytcnt**) Source File: (**fxs_src**) Destination File: (**fxs_dst**) Created at: (**fxs_ctime**)	

File Transfer Program Interface Error Codes		Part 3 of 14
Error Code	**Error Description**	**User Response**
282	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: There is not enough disk space available on the host. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	Delete some useless files on your host disk or ask the host system adminstrator for more disk space.
283	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: The request code was not correct. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	Retry the file transfer using correct request codes or use local problem-reporting procedures.
284	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: A specified flag is not correct, the specified host file does not exist on the host, or the host operating system is incorrect. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	Retry the file transfer using correct flags for the specified host operating system, retry the file transfer using an existing host file, or specify the correct host operating system.
285	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: There is an error reading from or writing to the host disk or partitioned data set specified is full. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	Check to make sure that the host disk you are transferring to or from exists, is valid, and can be written to or read from or increase the size of the specified partitioned data set.
286	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: The host option specified is invalid for this particular host operating system. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	Retry the file transfer specifying the correct host operating system or the correct flag value.

File Transfer Program Interface Error Codes		Part 4 of 14
Error Code	**Error Description**	**User Response**
287	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: Cannot write a file to the host. Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check to make sure that the host disk you are transferring to exists, is valid, and has write permissions.
288	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: Cannot read a file from the host. Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check to make sure that the host disk you are transferring from exists, is valid, and the file you are attempting to download from the host exists.
289	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: The TSO data set name specified is missing, already exists, or the host disk has read only access. Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Specify a data set name that is valid or talk with the host system administrator to get write permissions on the specified host disk.
290	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: The **–S** flag was specified incorrectly. Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Refer to the help menu by entering **fxfer –h** for the correct usage of the **–S** flag.
291	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: Do not specify the flags specified with a Partitioned Data Set. Source File: (**fxs_src**) Destination File: (**fxs_dst**)	

File Transfer Program Interface Error Codes		Part 5 of 14
Error Code	**Error Description**	**User Response**
292	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: Specify only one of tracks, cylinders, or avblocks when specifying the **–S** flag. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	
293	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: The CMS file identifier is missing or not correct. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	Specify all CMS file identifiers correctly.
294	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: The CMS file specified as the source file does not exist on the host. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	
295	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: The CMS disk has read permissions only. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	Talk with the host system administrator to get write permissions on the specified disk.

File Transfer Program Interface Error Codes		Part 6 of 14
Error Code	**Error Description**	**User Response**
296	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: Either the CMS disk is not accessible or the file already exists. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	Try to upload the file using the **–r** flag to replace the existing file on the host, or talk to the host system administrator.
297	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: The CMS disk is full. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	Delete some useless files on your host disk or ask the host system adminstrator for more disk space.
298	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: Invalid options specified without the **–a** flag. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	Use the **–a** flag to do the file transfer specified.
299	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: Specify the **–r** flag to replace the file. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	The specified file already exists on the host, the user must specify the **–r** flag to replace it.
300	Transfer Status: The file transfer completed with no errors. Byte Count: **(fxs_bytcnt)** Source File: **(fxs_src)** Destination File: **(fxs_dst)** Created at: **(fxs_ctime)**	

File Transfer Program Interface Error Codes		Part 7 of 14
Error Code	**Error Description**	**User Response**
301	Transfer Status: The file transfer request was cancelled by the host. Diagnostics: The host error message was not recognizable. Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check to make sure the host file transfer utility **(IND$FILE)** you are using is correct and up to date or use local problem reporting procedures.
302	**Warning:** The **dfxfer** file transfer daemon process attempt to send file transfer complete acknowledgement to the **fxfer** front end process failed. The **msgsnd()** system call error number is (**fxs_errno**).	The fxfer front end process was either removed or the emulator session was terminated.
303	Transfer Status: The file transfer request was cancelled. Diagnostics: A message sent by the **fxfer** front end process to the **dfxfer** file transfer background process failed. The **msgrcv()** system call error number is (**fxs_errno**).	Make sure that both the **fxfer** and **dfxfer** processes were not interrupted or use local problem reporting procedures.
304	Transfer Status: The file transfer request was cancelled. Diagnostics: Cannot read from the source file for uploading. The **read()** system call error number is (**fxs_errno**). Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check path name and permissions, try again later, or use local problem reporting procedures.
305	Transfer Status: The file transfer request was cancelled. Diagnostics: Cannot open the source file for uploading. The **open()** system call error number is (**fxs_errno**) Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check path name and permissions or use local problem reporting procedures.

File Transfer Program Interface Error Codes		Part 8 of 14
Error Code	**Error Description**	**User Response**
306	Transfer Status: The file transfer request was cancelled. Diagnostics: Cannot close the uploaded source file. The **close()** system call error number is (**fxs_errno**). Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check path name and permissions or use local problem reporting procedures.
307	Transfer Status: The file transfer request was cancelled. Diagnostics: The specified source file is empty. Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Files with zero bytes cannot be transferred.
309	Transfer Status: The file transfer request was cancelled. Diagnostics: Cannot write to the destination file. The **write()** system call error number is (**fxs_errno**). Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check path name and permissions, try again later or use local problem reporting procedures.
310	Transfer Status: The file transfer request was cancelled. Diagnostics: Cannot open the temporary replace file for downloading. The **open()** system call error number is (**fxs_errno**). Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check path name and permissions, do not use the **–r** flag, or use local problem reporting procedures.

File Transfer Program Interface Error Codes		Part 9 of 14
Error Code	**Error Description**	**User Response**
311	Transfer Status: The file transfer request was cancelled. Diagnostics: Cannot open the destination file for download. The **open()** system call error number is (**fxs_errno**). Source File: (**fxs_src**) Destinatino File: (**fxs_dst**)	Check path name and permissions or use local problem reporting procedures.
312	Transfer Status: The file transfer request was cancelled. Diagnostics: Cannot close the destination file. The **close()** system call error number is (**fxs_errno**). Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check path name and permissions or use local problem reporting procedures.
313	Transfer Status: The file transfer request was cancelled. Diagnostics: Cannot delete the original destination file. The **unlink()** system call error number is (**fxs_errno**). Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check path name and permissions, try to download to a different file name without using the **–r** flag, or use local problem reporting procedures.
314	Transfer Status: The file transfer request was cancelled. Diagnostics: Cannot link the **replace** file to the destination file. The **link()** system call error number is (**fxs_errno**). Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check path name and permissions, try to download to to a different file name without using the **–r** flag, or use local problem reporting procedures.

File Transfer Program Interface Error Codes		Part 10 of 14
Error Code	**Error Description**	**User Response**
315	Transfer Status: The file transfer completed with one warning. **Warning:** Cannot delete the temporary **replace** file name. The **unlink()** system call error number is (**fxs_errno**). Byte Count: (**fxs_bytcnt**) Source File: (**fxs_src**) Destination File: (**fxs_dst**) Created at: (**fxs_ctime**)	Check path name and permissions, try to download to a different file name without using the **–r** flag, or use local problem reporting procedures.
316	Diagnostics: Cannot open the diagnostics output file that contains the status information. The **open()** system call error number is (**fxs_errno**).	Check path name and permissions or use local problem reporting procedures.
318	Transfer Status: The file transfer request was cancelled. No **RESTART** file was created. Diagnostics: There was an error in the **RESTART** file input or output operation. A system call error occurred. The error number is (**fxs_errno**). Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check path name and permissions or use local problem reporting procedures.
319	Transfer Status: The file transfer request was cancelled. Diagnostics: Cannot log on to the host. The **AUTOLOG** or **LAF** program return code is (**fxs_errno**). Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check and make sure that the **LAF** or **AUTOLOG** script that is being used to log on to the host is correct by using the **tlaf** test tool program.

File Transfer Program Interface Error Codes		Part 11 of 14
Error Code	**Error Description**	**User Response**
320	Transfer Status: The file transfer completed with one warning. **Warning:** Could not get the destination file creation time. Byte Count: (**fxs_bytcnt**) Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check permissions or use local problem reporting procedures.
323	Transfer Status: The file transfer request was cancelled. Diagnostics: Cannot receive a message through the **fxfer** message queue. The **msgrcv()** system call error number is (**fxs_errno**). Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check to make sure that the HCON resource manager has not been removed, check that the **fxfer** and/or **dfxfer** processes have not been removed, check that the message queue in question has not been removed or use local problem reporting procedures.
325	Transfer Status: The file transfer did not complete. Diagnostics: The connection with the host is lost and no recovery for an explicit file transfer is available. Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check that the coaxial cable is still connected or use local problem reporting procedures.
326	Transfer Status: The file transfer request was cancelled. Diagnostics: Cannot send a message through the **fxfer** message queue. The **msgsnd()** system call error number is (**fxs_errno**). Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check to make sure that the HCON resource manager has not been removed, check that the **fxfer** and/or **dfxfer** processes have not been removed, check that the message queue in question has not been removed or use local problem reporting procedures.

File Transfer Program Interface Error Codes		Part 12 of 14
Error Code	**Error Description**	**User Response**
327	Transfer Status: The file transfer did not complete. Diagnostics: The connection with the host is lost. Cannot recover in the specified recovery time. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	Check that the coaxial cable is connected, check that the control unit is running, or use local problem reporting procedures.
330	Transfer Status: The file transfer request was cancelled. Diagnostics: The session that the user is attempting to recover is busy at this time. Source File: **(fxs_src)** Destinatino File: **(fxs_dst)**	Try again later or use local problem reporting procedures.
332	Transfer Status: The file transfer request was cancelled. Diagnostics: The response received from the emulator server is not recognized.	Log off the emulator server and try the file transfer again or use local problem reporting procedures.
333	Transfer Status: The file transfer request was cancelled. Diagnostics: The file transfer received an interrupt signal. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	A **RESTART** file should be created that will allow the user to restart the interrupted file transfer.
334	Transfer Status: An error occurred during the file transfer. Diagnostics: A system call failed. The system call error number is **(fxs_errno)**. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	Check path name and permissions, try the file transfer again or use local problem reporting procedures.

File Transfer Program Interface Error Codes		Part 13 of 14
Error Code	**Error Description**	**User Response**
338	Transfer Status: The file transfer request was cancelled. Diagnostics: Cannot get the status of the file transfer message queue. The **msgctl()** system call error number is (**fxs_errno**). Source File: (**fxs_src**) Destination File: (**fxs_dst**)	Check to make sure that the HCON resource manager has not been removed, check that the **fxfer** and/or **dfxfer** processes have not been removed, check that the message queue in question has not been removed or use local problem reporting procedures.
339	Transfer Status: The file transfer request was cancelled. Diagnostics: The **dfxfer** file transfer background process queue was removed. The system call error number is (**fxs_errno**). Source File: (**fxs_src**) Destination File: (**fxs_dst**)	The user either removed the emulator server session or the HCON resource manager before the file transfer had completed or before the file transfer wait time had run out.
340	Transfer Status: The file transfer request was cancelled. Diagnostics: An interrupt was received while the file transfer **fxfer** process was receiving a message from the **dfxfer** file transfer background process. The interrupt is not of a known type. The system call error number is (**fxs_errno**). Source File: (**fxs_src**) Destination File: (**fxs_dst**)	The user either removed the emulator server session or the HCON resource manager before the file transfer had completed or before the file transfer wait time had run out.
343	Restart Status: The **RESTART** file was created.	
344	Restart Status: Cannot create the **RESTART** file. There was an input or output error.	Check path name and permissions or use local problem reporting procedures.

File Transfer Program Interface Error Codes		Part 14 of 14
Error Code	**Error Description**	**User Response**
345	Transfer Status: The file transfer request was cancelled. Diagnostics: Cannot initialize the **e789x** emulator process. The **fork()** system call error number is **%1$d**.	Check for the **e789x** process in **/usr/bin**, check permissions, try again later, or use local problem reporting procedures.
347	Transfer Status: The file transfer request was cancelled. Diagnostics: The file transfer timed out. Source File: **(fxs_src)** Destination File: **(fxs_dst)**	If this is the user's first time to run a file transfer make sure that the **PSERVIC** value has been changed on the host and that **VTAM** has been recycled. Check to make sure that the host is set in extended mode, check that the control unit is not down, check that the host file transfer program variable within the session profile is valid, or use local problem reporting procedures.
349	The connection to the host is lost. Trying to recover the connection.	All file transfer requests are cancelled. Read through the AIX BOS manual on how to increase the system queue value or use local problem reporting procedures. Too many asynchronous file transfer requests may have been queued up also.
350	The system queue is full.	
352	A message sent to the dfxfer file transfer background process has been interrupted. The **dfxfer** file transfer background process may-have been deleted, the HCON resource manager may have been removed, or the file transfer message queue may have been removed.	Try the file transfer again or use local problem reporting procedures.

Implementation Specifics

The HCON File Transfer Program Interface error codes are part of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

Host file transfer program interface functions are the **fxfer** and **cfxfer** functions.

HCON AIX API Error Codes

The following table lists the valid AIX API error codes, describes the errors, and provides suggestions for user responses.

Summary of HCON AIX API Error Codes			page 1 of 8
Error Number & Name	**API Subroutine**	**Error Description & Reasons**	**User Response**
–1 G32_SESS_EXIST	alloc	A session exists on the logical path. The previous **g32_alloc()** function is not matched with a **g32_dealloc** function. or Failure on previous session's **g32_dealloc()** function: – missing **G32DALLOC** function in host application. – **as –> timeout** value too small.	Correct the API application appropriately.
–2 G32_NO_LA	open	There are no logical paths available. HCON device drivers not installed.	Wait for a logical path to become available. Check user profile.

Summary of HCON AIX API Error Codes			page 2 of 8
Error Number & Name	**API Subroutine**	**Error Description & Reasons**	**User Response**
–3 G32_EOPEN	open openx	An error occurred opening the logical path. **xerrinfo** contains the value of **errno** after the **open** system call: EINTR – the process was cancelled by the user. EIO – the application is attempting to use a link address that is being used by another system. EIO – possible hardware problem or link is down. Others	No user response required. Use correct link address. Check control unit. Contact your local IBM representative.
–5 G32_NO_LOG	close open	An error occurred while attempting implicit logon or logoff from the host. **xerrinfo** contains the return code from the **g32_logon** or **g32_logoff** programmatic logon routines.	Modify your **laf** script or AUTOLOG profile and test it with the **tlaf** utility.
–6 G32_NO_LP	open close alloc dealloc get_cursor get_data get_stat notify search send_keys read write	There are no free logical path IDs. The specified logical path does not exist: – missing **g32_open()** function. – failure on **g32_open()** function. or The logical path ID is invalid: **– as –> lpid** was incorrectly overwritten by the user program.	No user response required. Correct the API application appropriately.

Summary of HCON AIX API Error Codes			page 3 of 8
Error Number & Name	**API Subroutine**	**Error Description & Reasons**	**User Response**
–7 G32_NO_SESS	dealloc get_cursor get_data get_stat notify read search send_keys write	Missing or failure of the **g32_alloc()** function.	Correct the API application appropriately.
–8 G32_EEMU	open	The API could not start **e789** while executing implicit logging. The **xerrinfo** field contains:	
		≥0 e789x returned an error code.	Refer to the **$HOME/hconerrors** file for the emulator error.
		–1 Unexpected Interrupt	Reinstall HCON and try again.
		–2 **e789x** not found	Reinstall HCON and try again.
–9 G32_EMALLOC	alloc close dealloc get_cursor get_data read search send_keys write	The API was unable to allocate memory.	Record the information and contact your local IBM representative.
–10 G32_EFORK	open	The **fork** system call failed while starting an emulator. The **xerrinfo** field contains the value of **errno** returned from the **fork** system call.	Record the information and contact your local IBM representative.
–12 G32_ENDSESS	read write	The host application wishes to end. The API application is out of synchronization or host application failed.	Correct the API application appropriately.

Summary of HCON AIX API Error Codes			page 4 of 8
Error Number & Name	**API Subroutine**	**Error Description & Reasons**	**User Response**
–13 G32_INV_MODE		The specified mode is invalid for the particular function.	Correct the API application appropriately
	alloc	Only modes MODE_3270, MODE_API, and MODE_API_T are valid.	
	close	The application has not issued a **g32_dealloc()** function.	
	read get_stat write	Only modes MODE_API and MODE_API_T are valid.	
	get_cursor get_dat notify search send_keys	Only mode MODE_3270 is valid.	
–14 G32_MIX_MODE	alloc	The API mode is API/3270, but host program is API/API application.	Correct the API application appropriately.
–15 G32_PARMERR	alloc	The mode is API or API_T, but no host application name has been provided.	Correct the API application appropriately.
	open openx	The user did not supply a logon ID string or host password when performing implicit logging.	Correct the API application or input correct parameters when logging.
–16 G32_LINK_CTL	alloc	The API was unable to get control of the logical path.	Cancel and restart all HCON applications.

Summary of HCON AIX API Error Codes			page 5 of 8
Error Number & Name	**API Subroutine**	**Error Description & Reasons**	**User Response**
–17 G32_EREAD	alloc dealloc read write	The API received an error while attempting to read from the logical path.	
		The **xerrinfo** field contains the value of **errno** from the **read** system call:	
		EIO	Call the **g32_get_status** function. Match the value of **xerrinfo** with the emulator program error codes.
		EINTR	Cancel and restart all HCON applications.
		Other	Record the information and contact your local IBM representative.
		The **xerrinfo** field contains the value of **errno** from the **write** system call:	
		EIO	Call the **g32_get_status** function. Match the value of **xerrinfo** with the emulator program error codes.
		EINTR	Cancel and restart all HCON applications.
		Other	Record the information and contact your local IBM representative.
–18 G32_EWRITE	alloc dealloc read write	The API received an error while trying to write to the logical path.	Correct the API application appropriately.
–19 G32_ELENGTH	write	Invalid buffer size specified for the **g32_write()** function: – greater than **as –> maxbuf** – 0 or negative value	Correct the API application appropriately.

Summary of HCON AIX API Error Codes			page 6 of 8
Error Number & Name	**API Subroutine**	**Error Description & Reasons**	**User Response**
–20 G32_INV_POSITION	get_data search	The row or column specification is invalid. The application issued locations that are beyond the size of the presentation space.	Correct the API application appropriately.
–21 G32_INV_PATTERN	search	The specified pattern is invalid. The **xerrinfo** field contains the offset into the string of the invalid part of the pattern. Application does not follow the rules specified for the **g32_search** subroutine patterns.	Correct the API application appropriately.
–23 G32_SEARCH_FAIL	search	The target string was not in the presentation space.	
–24 G32_EMSGSND	open alloc close dealloc get_cursor get_dat notify open read search send_keys write	The API was unable to communicate with the emulator via the **msgsnd** system call. The **xerrinfo** field contains the **errno** from the **msgsnd** system call.	Refer to the **msgsnd** system call.
–25 G32_EMSGRCV	alloc close dealloc get_cursor get_dat open read search write	The API was unable to communicate with the emulator via the **msgrcv** system call. The **xerrinfo** field contains the **errno** from the **msgrcv** system call.	Refer to the **msgrcv** system call.

Summary of HCON AIX API Error Codes			page 7 of 8
Error Number & Name	**API Subroutine**	**Error Description & Reasons**	**User Response**
–29 G32_PROMPT	open	The API was unable to read the user ID and password during an implicit logon. This condition can occur if the AIX application was interrupted during prompting for user ID and password. (**Ctrl–Brk**).	Record the information and contact your local IBM representative.
–30 G32_EIOCTL	get_stat	Error occurred on **ioctl** system call. The **xerrinfo** field contains the **errno** returned by **ioctl** system call.	Record the information and contact your local IBM representative.
–31 G32_ESELECT	alloc dealloc read write	The API encountered an error while issuing the **poll** system call. The **xerrinfo** field contains the value of **errno**, which is generated by the **poll** system call.	Record the information and contact your local IBM representative.
–32 G32_NOTACK	dealloc read write	The API application is out of synchronization: – a **g32_read** does not match with a G32WRITE. – a **g32_write** does not match with a G32READ. – a **g32_dealloc** does not match with a G32DLLOC.	Correct the API application appropriately.
–33 G32_TIMEOUT	alloc dealloc read write	The timeout value was reached while waiting for data from the host. API application is out of synchronization.	Correct the API application appropriately.
		The timeout value was too small.	Increment the value of **as –> timeout**.
		An error has occurred on the host side of API.	Record the information from the workstation and the host, and contact your local system programmer or IBM.

Summary of HCON AIX API Error Codes			page 8 of 8
Error Number & Name	**API Subroutine**	**Error Description & Reasons**	**User Response**
–34 G32_OPENERROR	g32_open g32_openx	An error occurred while attempting to open multiple sessions. Only one session may be open at a time.	Issue a g32_close function before attempting the second open operation.

Implementation Specifics

The HCON AIX API error codes are part of the AIX 3270 Host Connection Program/6000 (HCON).

The HCON AIX API error codes are not available for Japanese Language Support.

Related Information

HCON AIX API Functions are the **g32_alloc** function, **g32_close** function, **g32_dealloc** function, **g32_fxfer** function, **g32_get_cursor** function, **g32_get_data** function, **g32_get_status** function, **g32_notify** function, **g32_open** function, **g32_openx** function, **g32_read** function, **g32_search** function, **g32_send_keys** function, and **g32_write** function.

HCON Host API Errors

The following table lists the Host API error codes, describes the errors, and provides suggestions for user responses.

Summary of Host API Errors			Part 1of 2
Error Number & Name	**API Subroutine**	**Error Description & Reasons**	**User Response**
–1 G32ESESS	G32READ G32WRITE G32DLLOC	An API session does not exist for this application	The host application must issue a **G32ALLOC** function before issuing a **G32READ** function, **G32WRITE** function, or **G32DLLOC** function.
	G32ALLOC	An API session already exists for this application.	The host application should issue a **G32DLLOC** function. The host application source code should be checked for multiple **G32ALLOC** functions.
–2 G32ESYS	G32ALLOC G32READ G32WRITE G32DLLOC	A host operating system error occurred while attempting to read or write. R0 contains one of the return codes listed in this table.	The host application should print the error code stored in R0 to the screen and terminate. Record the information on the emulator screen, contact your local host system programmer.
–3 G32ELEN	G32WRITE	The host API application attempted to send a message to the AIX API application, through the **G32WRITE** function, that was longer than the value returned from the **G32ALLOC** function, the length is negative, or the length was not defined.	Adjust the length sent to the **G32WRITE** function appropriately
–4 G32ETERM	G32READ G32WRITE	The AIX API application wishes to end the session or the API applications are out of synchronization.	Correct the API application(s) so the program flow is synchronized.

Summary of Host API Errors			Part 2 of 2
Error Number & Name	**API Subroutine**	**Error Description & Reasons**	**User Response**
–5 G32ENAPI	G32READ	Non–API message received during the **G32READ** routine on the host.	Contact your local IBM representative if the error occurs frequently.
–6 G32EMEM	G32READ	Error obtaining memory.	The host application should terminate. Record the information on the emulator screen and see your local system programmer.

Host API System Errors – VM/CMS

The Host API System Errors table lists system error codes, error names, and describes the error condition for a VM/CMS operating system.

Host API System Errors – VM/CMS Operating System		
Error Number	**Error Name**	**Error Description**
4	G32INCP	Console is in CP mode.
8	G32DISC	Console is disconnected.
12	G32INOP	Console is not operational.
16	G32UEX	Unit exception.
20	G32UCHK	Unit check.
24	G32CERR	Channel error.
28	G32INVD	Invalid device address.
32	G32INVC	Invalid command code.
36	G32NDWD	Parameter list not dword aligned.
40	G32INVR	Invalid interrupt routine address.
44	G32REDI	Display re–dialed.

Host API System Errors – MVS/TSO

The Host API System Errors table lists system error codes, error names, and describes the error condition for a MVS/TSO operating system.

Host API System Errors – MVS/TSO Operating System		
Error Number	**MVS/TSO Function**	**Error Description**
0	TGET	The data received by the terminal was edited by the **TGET** macro.
8	TPUT, TGET	An attention interruption occurred while the **SVC** routine for **TPUT/TGET** was processing. The message was not sent/received.
16	TPUT, TGET	Invalid parameters were received by the **TPUT** or **TGET SVC** routine.
20	TPUT, TGET	The host terminal was disconnected and could not be reached.

Implementation Specifics

The HCON Host error codes are part of the AIX 3270 Host Connection Program/6000 (HCON).

Related Information

Host API Functions are the **G32ALLOC** function, **G32DLLOC** function, **G32READ** function, and **G32WRITE** function.

cname Example VM/CMS File Transfer Program

The following program illustrates the file transfer process using the file transfer program interface on VM/CMS. To upload the file to an MVS/TSO host, the host option flag FXC_CMS must be changed to FXC_TSO and the file name must be in MVS/TSO format.

```
/************************************************************/
Program Name : cname - transfers a file using the file
                       transfer Program Interface
Module Name  : cname.c
Description  : The following program uploads a file from AIX
               to the VM/CMS host.  The file is translated from
               ASCII to EBCDIC using the translation tables of the
               language specified in the session profile.
               The file overwrites the host file, if it exists.
               The transfer is performed via explicit logon;
               that is, the user must establish a host session
               by logging on to the host and running the
               application from the 3278/79 subshell.
/*************************************************************/


#include <stdio.h>
#include <sys/types.h>
#include <fxfer.h>
#include <memory.h>
#include <time.h>
main()
{
     char *malloc();
     void free();
     void exit();
     unsigned int sleep();
     struct fxc *xfer;
     struct fxs sxfer;
     timer_t tmptime;
     char *c_time;
     register int LOOP;  /* indicates status not available yet */

     LOOP = 0;
  xfer = (struct fxc *) malloc (1024);
     xfer->fxc_src = "samplefile";     /* RT file to be uploaded */
     xfer->fxc_dst = "test file a";    /* VM/CMS Host file name  */

     ****************************************************************
                       Set the transfer flags
     ****************************************************************
     xfer->fxc_opts.f_flags = 0;
     xfer->fxc_opts.f_flags |= FXC_UP;   /* Upload file flag  */
     xfer->fxc_opts.f_flags |= FXC_TNL;  /* Translate—EBCDIC */
     xfer->fxc_opts.f_flags |= FXC_REPL; /* Replace host file if
                                                   it exists */
     xfer->fxc_opts.f_flags |= FXC_QUEUE;/* Transfer file,    */
                                         /* asynchronously  */
     xfer->fxc_opts.f_flags |= FXC_VAR;  /* Variable host  */
                                         /*   record format */
```

```
    xfer->fxc_opts.f_flags |= FXC_CMS;  /* VM/CMS host */
    xfer->fxc_opts.f_logonid = 0;     /* Explicit  file transfer*/
    xfer->fxc_opts.f_lrecl = 132;     /* Logical record length  */

/* from subshell of an  already logged on emulator session */

    if ( fxfer(xfer,(void *)0) == 0) }  /* Send file transfer
                                                  request*/
      for (;;) }
         switch(cfxfer(&sxfer)) }         /* Check status */
           case -1:
                  LOOP = 0;
                  printf("Status file not available due to\n");
                  printf(" status file I/O error\n");
/* Check hcon errors in your HOME directory for possible cause */
                                                  /*  of error  */
                  break;
           case 0:
                  LOOP = 0;
                  printf("Source file:%s\n", sxfer.fxs_src);
                  printf("Destination file:%s\n", sxfer.fxs_dst);
                  printf("Byte Count:%d\n", sxfer.fxs_bytcnt);
                  printf("MSG No:%d\n", sxfer.fxs_stat);
                  printf("ERROR if any:%d\n", sxfer.fxs_errno);
                  printf ("Destination file creation time:%s\n",
                sxfer.fxs_ctime);
                  break;

           case -2:
                  LOOP = 1;
                  sleep(15);
                  printf("Status not available yet\n");
                  break;
          }
          if ( !LOOP  ) }
                break;
          }
    }
  }

  free(xfer);
  exit(1);
}
```

fname Example FORTRAN File Transfer Program

```
  SAMPLE PROGRAM "sample fname"
  THIS FORTRAN PROGRAM WILL DOWNLOAD THE TSO FILE test.file TO
  THE AIX TEXT FILE /tmp/testfile.  THE FILE WILL BE TRANSLATED
  AND REPLACED ON AIX IF IT ALREADY EXISTS.  WE WILL BE INVOKING
  THE FILE TRANSFER TO SESSION a.
        INTEGER FCFXFER
        EXTERNAL FCFXFER
        INTEGER FFXFER
        EXTERNAL FFXFER
        CHARACTER*60 SRC
        CHARACTER*60 DST
        CHARACTER*60 SRCF
        CHARACTER*60 DSTF
        CHARACTER*25 LOGID
        CHARACTER*20 COMM
        CHARACTER*60 TIME
        INTEGER BYTCNT,STAT,ERRNO
        INTEGER FLAGS,RECL,BLKSIZE,SPACE,INCR,UNIT,RC1,RC2
5       FORMAT("------------------------------------------------")
6       FORMAT("FXFER RETURN CODE  =",1X,I4)
7       FORMAT("CFXER RETURN CODE  =",1X,I4)
8       FORMAT("SOURCE FILE        =",1X,A)
9       FORMAT("DESTINATION FILE =",1X,A)
10      FORMAT("BYTE COUNT         =",1X,I10)
11      FORMAT("TIME               =",1X,A)
12      FORMAT("STAT               =",1X,I10)
13      FORMAT("ERRNO              =",1X,I10)
        DSTF = '/tmp/testfile'//CHAR(0)
        SRCF = 'test.file'//CHAR(0)
        LOGID = CHAR(0)
        SRC = CHAR(0)
        DST = CHAR(0)
        RECL = 0
        BLKSIZE = 0
        SPACE = 0
C  THESE FLAGS REPRESENT TSO(1024) + REPLACE(16) + TRANSLATE(4) + DO
WNLOAD(2)
        FLAGS = 1046
C  WE WANT TO RUN THE FILE TRANSFER TO SESSION a.
        COMM = 'a'//CHAR(0)
        RC1 = FFXFER(SRCF,DSTF,LOGID,FLAGS,RECL,BLKSIZE,SPACE,
     + INCR,UNIT,COMM)
        WRITE(6,6) RC1
        RC2 = FCFXFER(SRC,DST,BYTCNT,STAT,ERRNO,TIME)
        WRITE(6,7) RC2
        WRITE(6,5)
        WRITE(6,8) SRC
        WRITE(6,9) DST
        WRITE(6,10) BYTCNT
        WRITE(6,11) TIME
        WRITE(6,12) STAT
        WRITE(6,13) ERRNO
        WRITE(6,5)
22      STOP
23      END
```

pname Example Pascal File Transfer Program

```
{**************************************************************}
{*                                                            *}
{* This Pascal program uses the File transfer pfxfer function   *}
{* to upload the binary file bin_file to a TSO host.The program *}
{* will prompt the user for a session profile to do the file    *}
{* transfer and will use the AUTOLOG script SYStso2 to logon to *}
{* the host id johndoe with trace and time=10 being set.        *}
{*                                                            *}
{**************************************************************}
program pname;

const

%include /usr/include/fxconst.inc

type

%include /usr/include/fxfer.inc
%include /usr/include/fxhfile.inc

var

        strc2   : fxs;
        strc1   : fxc;
        source, destin: stringptr;
        profid  : stringptr;
        login   : stringptr;
        rtval1  : integer;
        rtval2  : integer;
```

```
begin
      new(source, 256);
      new(destin, 256);
      new(profid, 256);
      new(login, 102);
      source@ := '/bin_file';
      destin@ := 'binfile';
{   Prompt the user for the session profile to use                   }
      writeln('Enter a profile id: ');
      readln(profid@);
{   Set login Host user id to johndoe, Autolog profile to SYStso2  }
{   turn trace on, and set time to 10 seconds.                       }
      login@ := 'johndoe,tso2,trace,time=10';
      strc1.fxc_opts.f_logonid := login;
{   Set the options to upload to a TSO host
    }
      strc1.fxc_opts.f_flags := FXC_UP + FXC_TSO;
      strc1.fxc_src := source;
      strc1.fxc_dst := destin;
      strc1.fxc_opts.f_lrecl := 0;
      rtval1 := pfxfer(strc1,profid);
      writeln('pfxfer return   = ',rtval1);
      rtval2 := pcfxfer(strc2);
      writeln('pcfxfer return  = ',rtval2);
      writeln('————————————————————————————————————————————————');
      writeln('Source file      = ',strc2.fxs_src@);
      writeln('Destination file = ',strc2.fxs_dst@);
      writeln('Byte count       = ',strc2.fxs_bytcnt);
      writeln('Time             = ',strc2.fxs_ctime@);
      writeln('Stats            = ',strc2.fxs_stat);
      writeln('Errno            = ',strc2.fxs_errno);
      writeln('————————————————————————————————————————————————');
      writeln;
end.
```

g32_sampl Example Program

The **g32_sampl** program is an example of an API application that tests API performance and verifies the implicit logon procedures.

The executable code is linked with the AUTOLOG procedure to allow the program to be executed implicitly or explicitly.

The **g32_sampl** program execution can take from ten minutes to one hour, and a control window allows the user to monitor the progress of data transmission between AIX and host systems. The window shows the buffer number in process, the maximum number of buffers for a particular session, and the direction of message transmission.

Procedure

Use the following commands to compile the **g32_sampl** program:

1. On AIX Workstations, use the following:
   - For standard workstations, enter:
     ```
     cc -c g32_sampl.c
     ```
   - For AIX Workstation with AUTOLOG option, enter:
     ```
     apilaf AUTOLOG g32_sampl g32_sampl.o -lcur -lcurses
     ```
   - For AIX Workstation with LAF script **/usr/lib/hcon/g_log.mvs**, enter:
     ```
     apilaf /usr/lib/hcon/g_log.mvs g32_sampl g32_sampl.o
     ```
2. On the Host System, use the following:
   - For VM/CMS, enter:
     ```
     g32asm g32sampl
     ```
   - For MVS/TSO, enter:
     ```
     exec g32asm 'g32sampl'
     ```

Inputs to g32_sampl

The **g32_sampl** program requires the following input:

- For implicit execution with the AUTOLOG option:

AUTOLOG_profile	In user's home directory or the **/usr/lib/hcon** file
Host user ID	First argument in user logon string
Host node ID	Second argument in user logon string
Host password	Response to the password prompt

- For implicit execution with LAF option:

  LAF_script
  Host user ID
  Host password

  No inputs are required for explicit execution.

Outputs from g32_sampl

- On AIX, the **g32_sampl** program displays a table that shows the mode, session, buffer information, and transmission direction.

  G32SAMPL evaluates HCON API performance for memory–to–memory transmission of one–megabyte data. Transmission time is a function of the session mode, length of the data stream, and direction.

- On VM/CMS host, the **g32sampl** program displays the upload and download time for the last session (only for explicit logon).
- On MVS/TSO host, the **g32sampl** program displays the following message: `END OF G32SAMPL PROGRAM` (only for explicit logon).

Execution

The **g32_sampl** program execution considerations are as follows:

- Run time option

  If no option is specified, the program uses the curses library to create the control window.

  If the **–c** option is specified, the program uses the standard output, and the control window is not created. Use the **–c** option if the output from **g32_sampl** is being redirected to a file.

- To execute the **g32_sampl** program using the implicit method (**g32_sampl** is linked with AUTOLOG):

  1. Start the application on AIX by entering:

     ```
     g32_sampl [–c]
     ```

  2. Enter the proper host logon ID string and password when prompted.

     The AUTOLOG or LAF **g32_logon** function logs on to the host.

     Upon completion of **g32_sampl**, the AUTOLOG or LAF **g32_logoff** function logs off from the host.

- To execute the **g32_sampl** using the explicit method:
  1. Establish the emulator session by entering `e789`.
  2. Log on to the host.
  3. Start an AIX subshell by using the emulator SHELL key (default is Ctrl–C).
  4. Run the sample program by entering:

     ```
     g32_sampl [–c]
     ```

     The **g32_sampl** program transfers data between the AIX and the host system, monitoring progress on the display.
  5. Upon completion of the **g32_sampl** program, return to the host by entering the `Ctrl–D` or `Exit` key sequence.
  6. End the HCON terminal emulation session by logging off and entering the `Ctrl–D` key sequence.

Related Information

Understanding Explicit and Implicit Logon on page 4–17, Understanding the Logon Assist Feature (LAF) on page 4–20, Understanding the HCON API on page 4–8, Understanding the File Transfer Program Interface on page 4–4.

g32_test Example Program

The **g32_test** program verifies the functional ability of the API library after installation of the host API library by the **instalapi** program.

The **g32_test** program also demonstrates how an application interfaces with the API library when an implicit logon is not needed. The user must add two dummy functions, **g32_logon** and **g32_logoff**, to the API application if the application will not be linked to **autolog.o** or **laf.o**.

Outputs from g32_test

On AIX, the **g32_test** program displays the following messages:

- Upon successful completion:

  `HCON HOST API INSTALLED AND OPERATIONAL`

- Upon unsuccessful completion:

  `The HCON host API is not functional`

Execution

The **g32_test** can only be executed explicitly. If the **g32_test** program is executed implicitly, the **g32_open** function returns the following error:

```
errcode = –5 (G32_NO_LOG)
```

```
xerrinfor = –2
```

Related Information

Understanding the HCON API on page 4–8, Understanding Explicit and Implicit Logon on page 4–17, Understanding AUTOLOG on page 4–19, Understanding the Automatic Logon Utilities on page 4–25.

The **g32_open** function.

g32_3270 Example Program

The **g32_3270** program demonstrates how to use the API/3270 subroutines. The **g32_3270** program performs the following functions:

- Uploads a host program called **g323270.**
- Assembles the **g323270** program.
- Executes the **g323270** program.
- Outlines the output from the **g323270** program in ** (asterisks).
- Sends portions of the 3270 presentation space to the screen.

Procedure

Use the following commands to compile the **g32_3270** program:

- On the AIX Workstation, enter:

  ```
  cc -a -o g32_3270.o
  ```

  - To link with the AUTOLOG option:

    ```
    apilaf AUTOLOG g32_3270 g32_3270.o
    ```

  - To link with the **/usr/lib/hcon/g_log.mvs** LAF script:

    ```
    apilaf /usr/lib/hcon/g_log.mvs g32_3270 g32_3270.o
    ```

Inputs to g32_3270

The **g32_3270** program requires the following input:

- For implicit execution with the AUTOLOG option:

  AUTOLOG_profile In user's home directory or the **/usr/lib/hcon** file

  Host user ID First argument in user logon string

  Host node ID Second argument in user logon string

  Host password Response to the password prompt.

- For implicit execution with LAF option:

  laf_script

  Host user ID

  Host password

  No inputs are required for explicit execution.

Outputs from g32_3270

On AIX the **g32_3270** program displays the following messages:

```
Uploading g323270 assemble
Assembling g323270 assemble
Executing g323270 on host
Outlining g323270 output

* * * * * * * * * * * * *
*                       *
     Greetings from
*                       *
          IBM
*                       *
      Austin, Texas
*                       *

* * * * * * * * * * * * *
```

Execution

- To execute the **g32_3270** program using the implicit method (**g32_3270.o** is linked with AUTOLOG)
  1. Start the application on AIX by entering:

     ```
     g32_3270
     ```

  2. Enter the proper host logon ID string and password at the prompt.

The AUTOLOG or LAF **g32_logon** function logs on to the host.

The **g32_3270** program transfers data between AIX and the host system. The program displays progress messages during the transfer process.

Upon completion of the **g32_3270** program, the AUTOLOG or Logon Assist Feature **g32_logoff** function logs off from the host.

- To execute the **g32_3270** using the explicit method:
  1. Establish the emulator session by entering `e789`.
  2. Log on to the host.
  3. Start an AIX subshell by using the emulator SHELL key (default is Ctrl-C).
  4. Run the API/3270 program by entering:

     ```
     g32_3270
     ```

The **g32_3270** program transfers data between the AIX and the host system. The program displays progress messages during the transfer.

  5. Upon completion of the **g32_3270** program, return to the host by entering `Ctrl–D` or `Exit`.
  6. End the HCON terminal emulation session by logging off and entering `Ctrl–D`.

Related Information

Understanding the HCON API on page 4–8, Understanding Explicit and Implicit Logon on page 4–17, Understanding AUTOLOG on page 4–19, Understanding the Automatic Logon Utilities on page 4–25, **g32_open** function

How To Install the HCON MVS/TSO Host API

The following procedure describes how to install the HCON MVS/TSO Host API.

Prerequisite Tasks or Conditions

Before the API can be installed on MVS/TSO, the following procedures must be completed:

1. The 3270 Connection Adapter or System/370 Host Interface Adapter must be installed.
2. The IBM 3270 File Transfer Program (**IND$FILE**) must be installed on the System/370 host.
3. HCON and HCON MRI must be installed.
4. You must have a user ID on the host system.
5. You must be an HCON user and have at least one session configured.

Procedure

Perform the following steps to complete the installation of the HCON API library on MVS/TSO:

1. Set your current working directory to **/usr/lib/hcon/mvs** by entering:

   ```
   cd /usr/lib/hcon/mvs
   ```

2. Establish an HCON emulation session on the console using the **e789** command.
3. Log on to the host. Process all output until there is no host activity and the screen is clear.
4. Start an AIX subshell by entering the emulator SHELL key (the default is `Ctrl–C`).
5. Start API installation by entering:

   ```
   instalapi
   ```

The installation program transfers the following files:

AIX Source Files	Host Destination Files
g32catal.cli	G32CATAL.CLIST
g32asm.cli	G32ASM.CLIST
g32alloc.mac	G32API.MACLIB.ASM(G32ALLOC)
g32data.mac	G32API.MACLIB.ASM(G32DATA)
g32dlloc.mac	G32API.MACLIB.ASM(G32DLLOC)
g32read.mac	G32API.MAXLIB.ASM(G32READ)
g32stat.mac	G32API.MAXLIB.ASM(G32STAT)
g32write.mac	G32API.MAXLIB.ASM(G32WRITE)
g32alloc.txt	G32API.TXTLIB.LOAD(G32ALLOC)
g32data.txt	G32API.TXTLIB.LOAD(G32DATA)
g32dlloc.txt	G32API.TXTLIB.LOAD(G32DLLOC)
g32read.txt	G32API.TXTLIB.LOAD(G32READ)
g32stat.txt	G32API.TXTLIB.LOAD(G32STAT)
g32write.txt	G32API.TXTLIB.LOAD(G32WRITE)
g32sampl.asm	G32APPL.ASM(G32SAMPL)
g32test.asm	G32APPL.ASM(G32TEST)

The installation program displays a status message as each file transfer completes. If the file transfer fails, the installation procedure terminates. The following example illustrates the messages returned as a result of a successful transfer:

```
0789—300   fxfer: Transfer Status:   The file transfer completed
                                     with no errors.

                  Byte Count:        1945
                  Source File:       /usr/lib/hcon/mvs/g32catal.cli
                  Destination File:  g32catal.clist
                  Created at:        (destination file create time)
```

After completing the file transfers, the **instalapi** program does the following:

- Compiles the **g32_sampl** and **g32_test** file.
- Catalogs the HCON API library in the user catalog.

When the installation process completes, the installation program displays the following message:

```
Installation of the HCON API library is complete.
```

6. Return to the emulator session by entering `Ctrl—D`.
7. The API library functions can be verified by entering the emulator SHELL key (the default is `Ctrl—C`) to start an AIX subshell.
8. Start the AIX API test program by entering:

   ```
   g32_test
   ```

   The **g32_test** program displays the message:

   ```
   HCON HOST API INSTALLED AND OPERATIONAL
   ```

   Note: If this message fails to appear, follow your local procedure for reporting problems.

Related Information

Understanding the Host Interface for HCON API on page 4–15, How To Install the HCON VM/CMS Host API on page 4–66.

How To Install the HCON VM/CMS Host API

The following procedure describes how to install the HCON VM/CMS Host Application Program Interface (API).

Prerequisite Tasks or Conditions

Before the API can be installed on VM/CMS, the following procedures must be completed:

1. The 3270 Connection Adapter or System/370 Host Interface Adapter must be installed.
2. The IBM 3270 File Transfer Program (**IND$FILE**) must be installed on the System/370 host.
3. HCON and HCON MRI must be installed.
4. You must have a user ID on the host system.
5. You must be an HCON user and have at least one session configured.

Procedure

Perform the following steps to complete the installation of the HCON API library on VM/CMS:

1. Set your current working directory to **/usr/lib/hcon/vm** by entering the following:

   ```
   cd /usr/lib/hcon/vm
   ```

2. Establish an HCON emulation session on the console using the **e789** command.
3. Log on to the host. Process all output until there is no host activity and the screen is clear.
4. Start an AIX subshell by entering the emulator SHELL key (`Crtl–C` is the default).
5. Start API installation by entering:

   ```
   instalapi
   ```

The **instalapi** program uploads the API files. It transfers the following files:

AIX Source Files	Host Destination Files
dispax.txt	DISPAX TEXT A
dispio.txt	DISPIO TEXT A
g32api.mac	G32API MACLIB A
g32api.txl	G32API TXTLIB A
g32sampl.asm	G32SAMPL ASSEMBLE A
g32test.asm	G32TEST ASSEMBLE A
g32asm.exe	G32ASM EXEC A

The installation program displays a status message as each file transfer completes. An example of a successful transfer message is as follows:

```
0789–300 fxfer:   Transfer Status:    The file transfer completed
                                       with no errors.
                  Byte Count:          12080
                  Source File:        /usr/lib/hcon/vm/g32api.mac
                  Destination File:    G32API MACLIB
                  Created at:         (destination file create time)
```

The **instalapi** program also:

- Creates the **DISPAX** and **DISPIO** module files.
- Compiles the **G32SAMPL** and **G32TEST** files.

The installation program indicates successful installation of the API library by displaying the following message:

```
Installation of the HCON API library is complete.
```

6. Return to the emulator sesion by entering `Ctrl–D`.
7. To verify the API functions, start an AIX subshell by entering the emulator SHELL key, (`Ctrl–C` is the default).
8. Start the AIX API test program by entering:

   ```
   g32_test
   ```

   The **g32_test** program displays the message:

   ```
   HCON HOST API INSTALLED AND OPERATIONAL
   ```

   Note: If this message fails to appear, follow your local procedure for reporting problems.

Related Information

Understanding the Host Interface for HCON API on page 4–15, How To Install the HCON MVS/TSO Host API on page 4–64.

How To Compile a File Transfer Program

The following procedure describes how to compile a file transfer program.

Prerequisite Tasks or Conditions

1. HCON and HCON MRI must be installed and configured on your system.
2. The specific computer language compiler, either Pascal, FORTRAN, or C, in which the file transfer program was written must be installed on your system.

Procedure

The following sections describe how to compile a file transfer program for each of the three languages, C, FORTRAN, and Pascal.

Compiling a C Program

The following example illustrates how to compile the sample **cname** C program and link it with the file transfer library:

```
cc -o cname cname.c -lfxfer
```

where **cname** is the C program executable code and **cname.c** is the actual C file transfer program. The C file transfer program is linked with the **libfxfer.a** library.

Compiling a FORTRAN Program

The following example illustrates how to compile the sample **fname** FORTRAN program and link it with the file transfer library:

```
xlf -v fname.f -o fname -lfxfer
```

where **fname.f** is the name of the FORTRAN file transfer program and **fname** is the name of the FORTRAN executable program.

Compiling a Pascal Program

The following example illustrates how to compile the sample **pname** Pascal program and link it with the file transfer library:

```
xlp -v pname.pas -o pname -l fxfer
```

where **pname.pas** is the Pascal file transfer program and **pname** is the Pascal executable code.

Related Information

Understanding the File Transfer Program Interface on page 4–4, Understanding HCON Programming Examples on page 4–25.

Understanding the File Transfer Process in *Communication Concepts and Procedures.*

How To Compile an AIX API Program

The following procedure describes how to compile an AIX API program.

Prerequisite Task or Condition

1. HCON must be installed and configured on the RISC System/6000.
2. You must have installed a Pascal, FORTRAN or C compiler.

Procedure

The following sections describe how to compile an AIX API program written in either the C, Pascal, or FORTRAN language.

Compiling a C Program

To compile a typical C HCON API application, such as **g32_3270.c**, enter:

```
cc –c g32_3270.c
```

Note: The **g32_3270.c** program is a sample program existing in the **/usr/lib/hcon/vm** directory and **/usr/lib/hcon/mvs** directory.

To link the **g32_3270** program with AUTOLOG, enter:

```
apilaf AUTOLOG g32_3270.c g32_3270.o
```

To compile a C HCON API application, enter:

```
cc –c testap.c
```

where testap.c is the name of the HCON API application you want to compile.

To link it with the HCON API application with the LAF script **/usr/lib/hcon/g_log.vm,** enter:

```
apilaf /usr/lib/hcon/g_log.vm testap testap.o <other required
                                       object files>
```

Compiling a Pascal Program

To compile a Pascal HCON API program, such as **testap.pas**, enter:

```
xlp –c testap.pas
```

To link a Pascal HCON API program, such as **testap.pas**, with AUTOLOG, enter:

```
apilaf –p AUTOLOG testap testap.o <other required object files>
```

Compiling a FORTRAN Program

To compile a FORTRAN HCON API program, **apiftest.f**, enter:

```
xlf –c apiftest.f
```

To link a FORTRAN HCON API program, such as **testap.pas**, with AUTOLOG, enter:

```
apilaf –f AUTOLOG apiftest apiftest.o <other required object files>
```

Related Information

Understanding the HCON Application Program Interface (API) on page 4–25, Understanding the AIX Interface for HCON API on page4–11 , Understanding HCON Programming Examples on page 4–25, Understanding AUTOLOG on page 4–19, Understanding the Logon Assist Feature (LAF) on page 4–20.

How To Use an AUTOLOG Profile on page 4–74, How to Use a Logon Assist Feature Script on page 4–71.

How To Compile a Host HCON API Program

The following procedure describes how to compile a Host HCON API program.

Prerequisite Tasks or Conditions

1. HCON and HCON MRI must be installed and configured on the RISC System/6000.
2. The HCON API programs must be installed on the host system using the **instalapi** program.
3. You must have a host ID.

Procedure

The following sections describe how to compile a 370 Assembler program in either of the Host environments, VM/CMS or MVS/TSO.

VM/CMS

The **g32asm exec** file is provided for compiling a HCON API application on a VM/CMS host.

To compile a 370 Assembler program called **apitest assemble** with the VM/CMS HCON API library, enter:

```
g32asm apitest
```

MVS/TSO

The **g32asm.clist** file is provided for compiling a HCON API application on a MVS/TSO host. To compile a 370 Assembler program called **apitest** with the MVS/TSO HCON API library, perform the following:

1. Place **apitest** into the `g32appl.asm` partitioned file.
2. Compile and link **apitest** with the HCON API by entering:

   ```
   exec g32asm 'apitest'
   ```

Note: The prefix **g32** is reserved for the HCON API library. This prefix must not be used for naming VM/CMS or MVS/TSO application programs.

Related Information

Understanding the HCON Application Program Interface (API) on page 4–4, Understanding the Host Interface for HCON API on page 4–15, Understanding HCON Programming Examples on page 4–25, Understanding AUTOLOG on page 4–19, Understanding the Logon Assist Feature (LAF) on page 4–20.

How To Install the HCON MVS/TSO Host API on page 4–64, How To Install the HCON VM/CMS Host API on page 4–66.

How To Use a Logon Assist Feature Script

HCON provides sample Logon Assist Feature (LAF) scripts to help you create, debug, and link files using the LAF language. The first procedure describes how to use the sample LAF scripts. The second procedure describes how to debug and link your own LAF script. The LAF language is not the only method of incorporating automatic logon into your applications. The AUTOLOG profile also provides this capability.

Prerequisite Tasks or Conditions

1. You must have installed HCON and HCON MRI.
2. You must have a C compiler installed on your system.
3. You must have written a LAF script using the LAF language statements.
4. You must have a host ID.
5. You must have a 3270 Connection Adapter or System/370 Host Interface Adapter.
6. You must be an HCON user and have established at least one session.

How to Use the Sample LAF Scripts

The steps below are provided to help you use the provided LAF script for creating, debugging, and linking files and to help you better understand the processes.

Procedure

1. Copy one of the example LAF scripts in **/usr/lib/hcon** into your home directory. For example:

   ```
   cp /usr/lib/hcon/g_log.vm g_log.vm
   ```

   or

   ```
   cp /usr/lib/hcon/g_log.mvs g_log.mvs
   ```

2. If you have the capability on your system, edit the copied LAF script while logging on and off of the emulator in another AIX shell. The changes should reflect your particular logon and logoff process. If you want **DEBUG**, then insert the **DEBUG** statement after the **START** statement or before the statements that you want **DEBUG** output for.
3. Given that **g_log.vm** is the file name of your LAF script, create a test program for your LAF script. To create the **tlaf** command executable test program, enter:

   ```
   mtlaf g_log.vm
   ```

When running the **tlaf** test program, you must have the **e789** program also running.

4. Run the emulator command with a session profile name. To run the **e789** command with a session profile of `z`, enter:

   ```
   e789 z
   ```

5. Execute the **tlaf** program, within the emulator subshell or within another shell. The command prompts you for the user ID string and password, unless the host login ID is specified in the session profile. If the login ID is specified, the program only prompts you for the password. Enter or create your logon ID and/or password. To execute a **tlaf** program with a session profile name of `z`, enter:

   ```
   tlaf z
   ```

   The LAF script attempts to log on and log off. You must change the LAF script if you do not get a successful logon and logoff. Continue to debug the LAF script until you get a successful logon and logoff. When you get a successful logon and logoff, you have a LAF script that is ready to test with an API program or the **fxfer** file transfer program interface.

6. Turn on the **DEBUG** so that you may run the **tlaf** command and debug your LAF script.

7. Upon successful completion of the **tlaf** program, link the HCON applications with your LAF script by entering one of the following link commands.

   For file transfer programs, use:

   ```
   fxlaf g_log.vm
   ```

   For API application programs, use:

   ```
   apilaf g_log.vm
   ```

Note: The **DEBUG** statement should be removed from the LAF script when linking with an API or **fxfer** application. It is not necessary to run the emulator when running an implicit API or **fxfer** application. If you are using LAF, the AUTOLOG Node ID in the session profile must be blank.

Another LAF program, which enables the user to log onto a MVS host, is also provided in the **/usr/lib/hcon** directory in the file **g_log.mvs**.

Testing a LAF Script

To test the LAF script with an API program, complete the following steps:

1. Compile the API program:

   ```
   cc -c apiprog.c
   ```

2. Now link the API program object with the LAF script object to create an API executable program:

   ```
   apilaf laf.script apiprog apiprog.o
   ```

3. Execute the API program without initiating an emulator session. This will cause the API program to prompt you for logon ID and a password, unless you have specified a host login ID in the session profile. If you have specified a login ID in the session profile, the program only prompts you for the password.

Occasionally, your script may not successfully log on or log off even though your **tlaf** program works. In this case you will have to keep adjusting or debugging the LAF script until it works. It is a good idea to look at the C program that is generated from the LAF script.

To test the LAF script with a file transfer application program, complete the following:

- Run the **fxfer** command with a LAF script by entering:

  ```
  fxlaf lafscript
  ```

  This command links the LAF script with the file transfer **dfxfer** process.

- If you are running in a codeserver environment, the following command must be run to link the LAF script with file transfer:

  ```
  apilaf <lafscript> dfxfer /usr/lib/hcon/dfxfer.o
  ```

  The **apilaf** command creates a new **dfxfer** program in your current working directory. You must enter the new location of the **dfxfer** program in the PATH environment variable before the **/usr/bin** entry, so that the newly created **dfxfer** is located and used before the system reads the original **dfxfer** program in the **/usr/bin** directory.

To generate a C program from a LAF script, execute the following program:

```
genlaf lafscript
```

The C program will be contained in the **laf.c** file.

Related Information

The **apilaf** command, **fxlaf** command, **mtlaf** command, **tlaf** command.

Understanding Explicit and Implicit Logon on page 4–17, Understanding the Logon Assist Feature (LAF) on page 4–20, Understanding the HCON API on page 4–8, Understanding the File Transfer Program Interface on page 4–4.

How To Use an AUTOLOG Profile

Using an AUTOLOG Profile

The following procedure describes how to create, test, link, and execute an AUTOLOG profile.

Prerequisite Tasks or Conditions

1. The 3270 Connection Adapter or System/370 Host Interface Adapter must be installed.
2. You must have HCON and HCON MRI installed on your system.
3. You must be an HCON user and have at least one session profile defined.
4. You must have a user ID on the host.
5. You must have written an AUTOLOG profile using the **genprof** command.

   Note: The **/usr/lib/hcon** directory provides sample AUTOLOG profiles, SYSvm1, SYSvm2, and SYStso. The directory also contains the **logform** file which shows what the format of the **genprof** command menu looks like.
6. The **genprof** command must have produced a **SYS***name* file (where *name* = the node id you specified when using the **genprof** command).
7. You must have a C compiler installed on your system to use the **mtlaf**, **fxlaf**, and **apilaf** programs.
8. You must have a **tlaf** program.
   a. If you are only working with AUTOLOG, use the existing **tlaf** program in the **/usr/lib/hcon** directory.
   b. If you have used a LAF script and want to create a new **tlaf** program, use the **mtlaf** command. To execute the **mtlaf** command, enter:

      ```
      mtlaf AUTOLOG
      ```

Procedure

The following steps describe how to create an AUTOLOG profile.

1. Run the **e789** emulator program. You must provide a specific session profile. To run **e789** with session profile `z`, you would enter the following:

   ```
   e789 z
   ```
2. Log on to the host, then log off. Record all the events on an AUTOLOG logform, until the final logon/logoff prompt.
3. Generate the log profile by running the **genprof** command. The **genprof** command offers menus that allow the user to create, display, or change an AUTOLOG profile. The **genprof** command stores the log information in the user's HOME directory. The log information resides in the log profile under the name, SYS*node_id*. For example:

   ```
   SYSdanvml
   ```

   The suffix `danvml` represents the node id.

At this point, you have two options: You can use the **tlaf** program provided in the **/usr/lib/hcon** directory or you can create a test program for AUTOLOG using the **mtlaf** command. The **mtlaf** command creates an executable **tlaf** program in your working directory. To use the existing **tlaf** program, skip step 4 and go to step 5.

Note: The **mtlaf** command requires a C compiler.

4. To create a test program for AUTOLOG using the **mtlaf** command, enter:

   ```
   mtlaf AUTOLOG
   ```

5. Run the test program using the **tlaf** command. You must specify the same session profile that you used to start the emulator in step 1. For example, to use the **tlaf** command, if you have used session profile z for the **e789** command, enter:

   ```
   tlaf z
   ```

   If the host login ID is not specified in the session profile, the **tlaf** command prompts you for the logon ID string and password. Otherwise, the program only prompts you for the password. Enter the logon ID string in the following format:

   ```
   uid,nid,trace,time=value
   ```

   For example:

   ```
   grace,danvm1,trace,time=5
   ```

   where:

   uid Specifies the User ID.

   nid Specifies the Node ID. The Node ID is name of the AUTOLOG profile (excluding the SYS prefix).

   trace Specifies optional control output.

   time Specifies optional parameter, which allows the user to change the three–second default time to wait for a particular pattern to be received from a host.

   value Specifies the time in seconds for AUTOLOG to wait for a particular pattern to be received from the host.

If the AUTOLOG procedure cannot find the AUTOLOG profile in your $HOME directory, it checks the **/usr/lib/hcon** directory. If the procedure cannot find the AUTOLOG profile in the **/usr/lib/hcon** directory, the AUTOLOG procedure displays an error message.

The **tlaf** program, located in the **/usr/lib/hcon** directory, is linked to AUTOLOG. At installation, the **dfxfer** file transfer subsystem is automatically linked with AUTOLOG. Therefore, with the exception of API programs, you do not need to do anything else to link HCON applications with the AUTOLOG procedure.

Testing the AUTOLOG Profile

If you are going to test the AUTOLOG profile with an API program, do the following:

1. Compile the API program:

   ```
   cc -c apiprog.c
   ```

2. Link the API program object with the **autolog.o** object file to create an API program executable. For example:

   ```
   apilaf AUTOLOG apiprog apiprog.o
   ```

3. Execute the API program without initiating an emulator session. If the host login ID is not in the session profile, the API program prompts you for a logon ID and a password. Otherwise, the program only prompts you for the password.

If you are going to test the AUTOLOG profile with the file transfer command or program, do the following:

- Run the file transfer command or program implicitly specifying the **-n** session name option.

Occasionally, your AUTOLOG profile may not successfully log on or log off even though your **tlaf** program works. In this case, keep adjusting or debugging the profile until it works.

When debugging, enable the tracing facility. By setting AUTOLOG trace on in the session profile or if the logon ID is prompted, simply enter in a comma and the word trace after the AUTOLOG profile *name*. For example:

```
Enter login id:
dummyid,node id,trace
```

If the host machine is slow during debugging, you may also want to use the time facility. The time option allows the user to specify the maximum amount of time in seconds to wait for a successful event.

To enable the time facility by setting the AUTOLOG *timeout* parameter in the session profile or if the logon ID is prompted, enter:

a. the AUTOLOG name and AUTOLOG trace value

b. a comma and time = *N seconds*

For example:

```
Enter login id:
dummyid, node id, trace, time=10
```

Linking to AUTOLOG

If you have used a Logon Assist Feature (LAF) script and want to switch to using AUTOLOG or have used a combination of both, you need to use additional commands to link to AUTOLOG.

The following commands link an HCON application with the AUTOLOG procedure:

fxlaf Links a file transfer program to AUTOLOG

apilaf Links an API application to AUTOLOG.

Prerequisite Task or Condition

- You must have a C compiler installed on your system.

Procedure

- To link a file transfer HCON application with the AUTOLOG procedure after the **tlaf** program runs successfully, enter:

```
fxlaf AUTOLOG
```

- To link an API HCON application program with the AUTOLOG procedure, enter:

```
apilaf AUTOLOG name obj.o
```

Related Information

The **apilaf** command, **fxlaf** command, **genprof** command, **mtlaf** command, **tlaf** command, **/usr/lib/hcon** directory.

Understanding the Logon Assist Feature (LAF) on page 4–17, Understanding the HCON API on page 4–8, Understanding the File Transfer Program Interface on page 4–4, Understanding Explicit and Implicit Logon on page 4–17, Understanding AUTOLOG on page 4–19, Understanding the Automatic Logon Utilities on page 4–25.

Understanding Emulator Sessions in *Communication Concepts and Procedures*, Installing and Updating HCON, Configuring HCON.

Chapter 5. Network Computing System (NCS)

The Network Computing System (NCS) enables the distribution of processing tasks across resources in a network or internet by maintaining databases that control the information about the resources. NCS consists of three components: the Remote Procedure Call runtime library, the Location Broker, and the Network Interface Definition Language compiler. This chapter provides detailed information on the working of NCS and its components. In addition, it provides brief introductions to the various library routines used in NCS.

NCS Overview

The Network Computing System (NCS) is a set of tools for distributing computer processing tasks across resources in either a network or several interconnected networks (an internet). NCS is an implementation of the Network Computing Architecture, which distributes software applications across networks and internets that include a variety of computers and programming environments. Programs based on the Network Computing Architecture take advantage of computing resources throughout a network or internet, with different parts of each program executing on the computers best suited for certain tasks.

The following figure shows how the Network Computing Architecture shares processing and application data:

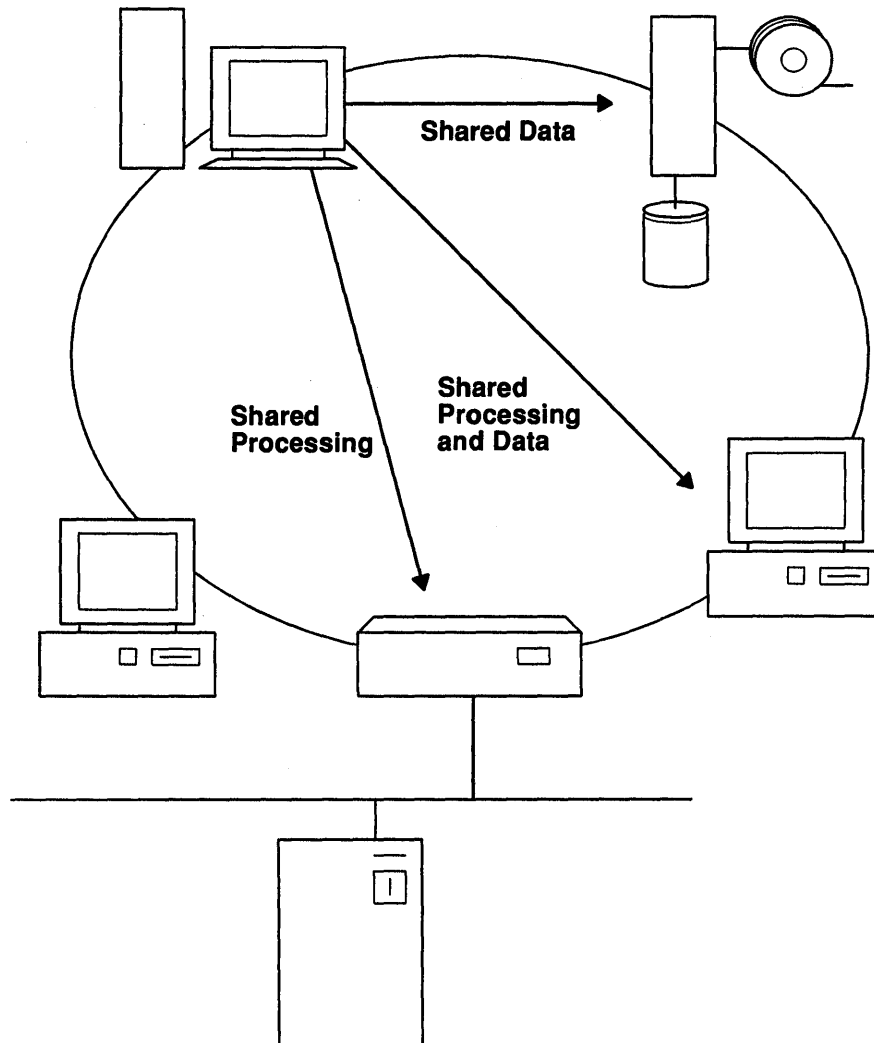


Figure 1. Distributed Computing Using NCS

Understanding NCS

NCS consists of the following components:

- Remote Procedure Call (RPC) runtime library
- Location Broker
- Network Interface Definition Language (NIDL) compiler.

The RPC runtime library and the Location Broker provide runtime support for network computing. Together these two components make up the Network Computing Kernel (NCK), which contains all the software required to run a distributed application. The NIDL compiler is a tool for developing applications.

RPC Runtime Library

The RPC runtime library provides the library routines that enable local programs to execute procedures on remote hosts. These routines transfer requests and responses between clients (the programs calling the procedures) and servers (the programs executing the procedures). When you write a distributed application, you usually do not need to use RPC routines directly. Instead, you can create an interface definition in Network Interface Definition Language and use the NIDL compiler to generate the required RPC routines.

NIDL Compiler

The NIDL compiler takes as input an interface definition written in NIDL. An interface definition specifies the interface between a user of a service and the provider of the service. It defines both the way in which a client application sees a remote service and the way in which a remote server sees requests for its service. From this definition, the NIDL compiler generates client and server stub source code and header files.

The client stub program performs the conversion between requests (and responses) that are meaningful to the client and packets that are transmitted (and received) on the network. The server stub program provides similar support for the server.

The stubs produced by the NIDL compiler contain nearly all of the remoteness in a distributed application. The stubs perform data conversions, assemble and disassemble packets, and interact with the RPC runtime library. It is much easier to write an interface definition in NIDL than it would be to write the stub code that the NIDL compiler generates from your definition.

Location Broker

The Location Broker provides information about the network or internet resources to clients. It maintains a database that contains the identities and locations of objects in the network. Through a Client Agent, the Location Broker maintains information about the local brokers that manage information about resources on the local host, the global brokers that manage information about resources available on all hosts, and the administrative tools.

Objects, Types, and Interfaces

Like the architecture on which it is based, NCS is object-oriented. An object is an entity accessed or manipulated by well-defined operations. Files, serial lines, printers, and processors can all be objects.

Programs access objects through interfaces. The programs are cast in terms of the objects they manipulate instead of the machines with which they communicate. Object-oriented programs are easy to design and can readily accommodate changes to hardware and network configurations.

Every object has a type that specifies the class or category of the object. All objects of a type are accessed through one or more interfaces. Each interface is a set of operations that can be applied to any of the objects of that type. For example, you can classify printer queues as objects of the type printqueue, which are accessed through a **printqueue_ops** interface that includes operations to add, delete, and list jobs in the queues.

The definition of an operation specifies its input and output parameters, but not its implementation. Therefore, an operation can be implemented differently on different types of objects.

Array processors provide an example of how objects, types, and interfaces apply to NCS. You can define an **arrayproc** type. Array processor objects are accessed through either of two interfaces: a **vector_ops** interface with operations such as **vector_add** and **vector_multiply**, and a **misc_ops** interface with operations such as **max_absolute_value** and **root_mean_square**.

UUIDs

NCS identifies every object, type, and interface by a Universal Unique Identifier (UUID). Each UUID is a 16-byte quantity identifying the host on which the UUID is created and the time at which it is created. Six bytes identify the time, two are reserved, and eight identify the host.

The Network Computing Kernel (NCK) includes a **uuid_gen** utility that generates UUIDs as ASCII strings or as data structures defined in the C or Pascal programming language. The string representation used by the NIDL compiler and by NCK utilities consists of 28 hexadecimal digits arranged as in this example:

```
3a2f883c4000.0d.00.00.fb.40.00.00.00
```

Clients and Servers

A client is a program that makes remote procedure calls. A remote procedure call requests that a particular operation be performed on a particular object. A server is a program that implements interfaces. The server process listens for requests for each interface's operations. When it receives a request from a client, the server executes the procedures that perform the operation and sends a response to the client.

Network Communications

The communications between systems in an NCS environment are handled through the RPC runtime library. It is possible that one program can access different hosts that listen on two different ports or have two different addresses.

In the NCS environment, RPC uses sockets for interprocess communications. A socket is an end point for communications, in the form of a message queue. An RPC server listens on one or more sockets. It receives any message sent to a socket on which it is listening. Messages can be broadcast to sockets at several hosts on the local network. Broadcasting is often used when the location of an object is not known.

The following figure illustrates RPC communications using sockets. It shows two servers running on one host and several clients on other hosts. Server 1 listens on two sockets: one socket uses the Internet Protocol (IP); the other uses another network communications interface. Server 2 listens on a socket that uses IP.

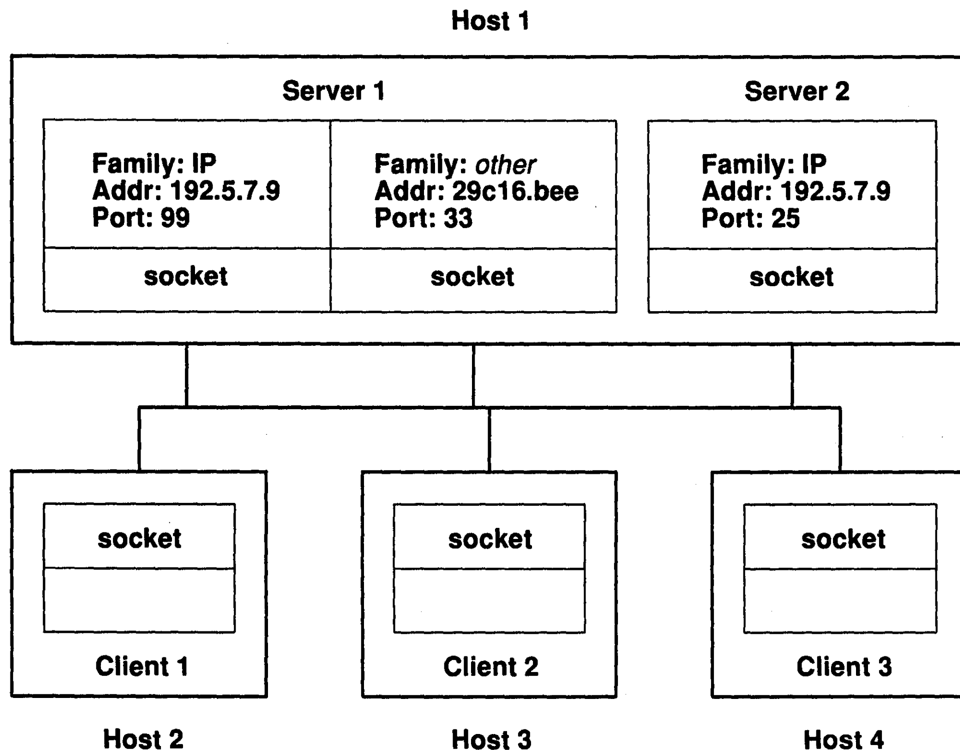


Figure 2. RPC Communications Using Sockets

Each socket is identified uniquely by a socket address. A socket address, sometimes called a sockaddr, is a data structure that specifies the following information about a socket:

- Address family, also called the protocol family, which determines the communications protocol used to deliver messages and the structure of the addresses used to represent communications end points.
- Network address, which is a value that, given the communications protocol, uniquely identifies a host on one or more interconnected networks.
- Port number, which specifies a communications end point within the host. The terms port and socket are synonymous, but port number and socket address are not. A port number is one of the three parts in a socket address. For example, a port number can be represented as the character string `77`, while a socket address can be represented as `ip:myhost[77]`.

The following figure illustrates the socket address structure for a domain socket address and an IP socket address.

Domain Socket Address

<table>
<tr><th rowspan="1">Family</th><th>Port</th><th colspan="2">Network Address</th></tr>
<tr><td rowspan="2">16-bit integer</td><td rowspan="2">16-bit integer</td><td>Network</td><td>Host</td></tr>
<tr><td>32-bit integer</td><td>32-bit integer</td></tr>
</table>

IP Socket Address

Family	Port	Network Address
16-bit integer	16-bit integer	32-bit integer

Figure 3. Domain Socket Address and IP Socket Address Structures

Well-Known and Opaque Ports

Interfaces can be designed and implemented with a particular port number *built in*. The port used in such an interface is called a well-known port. Clients of the interface always send to that port, and servers always listen on that port. Some well-known ports are assigned to particular servers by the administrators of a communications protocol. For example, the administrators of the Internet Protocols have assigned port number 23 to the server for the Telnet remote login facility. All Telnet servers listen on this well-known port, and all Telnet user programs send to it.

Well-known ports are an effective way to coordinate communication between clients and servers if portability to other networks and coexistence with other services are of little concern. However, the number of ports in each protocol family is limited. Unless the assignment is obtained from a central administrator, an application's well-known port number is liable to conflict with that of another program. The NCS Location Broker circumvents this problem by allowing you to locate services easily without direct use of well-known ports. It uses one well-known port to listen for requests. Clients and servers can locate a broker by broadcasting to this port.

NCS enables a server to use ports that the RPC runtime software assigns dynamically. After a server registers this assignment with the Location Broker, a client can then obtain the server's socket address from the broker. Since there is no need for either the client or the server to know a port number, the number is said to be opaque.

The RPC Paradigm

Remote procedure calls extend the procedure call mechanism from a single system to a distributed computing environment. The calls distribute the execution of a program among multiple computers in a way that is transparent to the application-level code.

The following figure shows the flow of ordinary local procedure calls between the calling client and the called procedures.

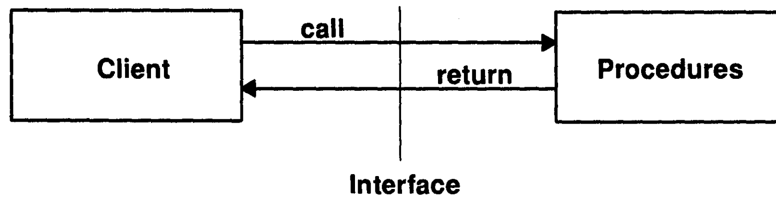


Figure 4. Single-Process Procedure Call Flow

The following figure shows the same flow for remote procedure calls and illustrates how RPC hides the remote aspects of a call from the calling client. The client application requests a procedure by using standard calling conventions, as if the procedure were a part of the local program, the procedure is, however, executed by a remote server. The client stub acts as the local representative of the procedure, organizing the data into a format that can be transmitted to the server and using RPC runtime library routines to communicate with the server. Similar activities occur within the server process.

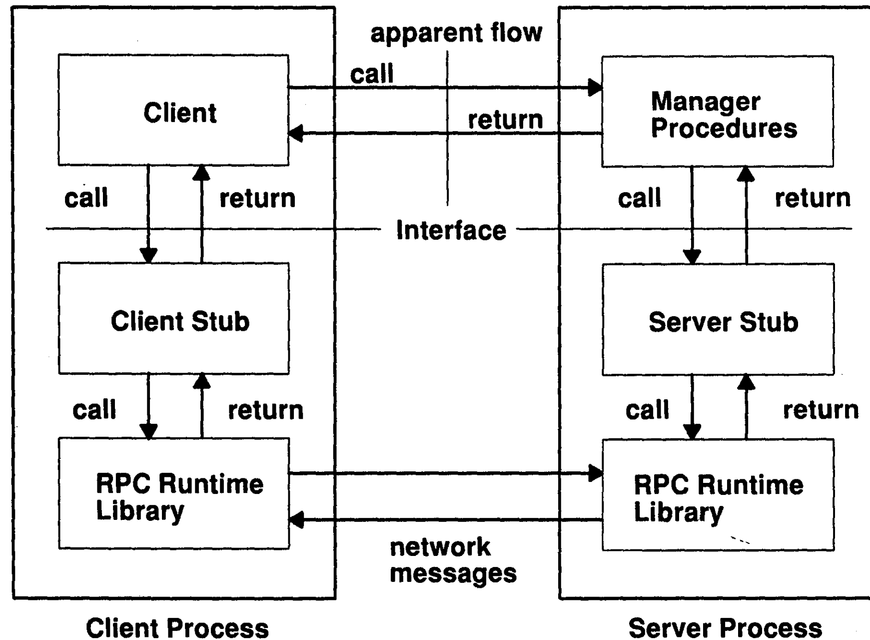


Figure 5. Remote Procedure Call Flow

Interfaces

An interface consists of procedure names and signatures. It defines the calling syntax that is used both by the client and by the remote procedures. An interface is independent of the mechanism that conveys the request between the client and the procedures; it is also independent of the way in which the procedures perform the operations. The server that implements an interface's operations is said to export the interface. The program that requests the operations imports the interface.

For example, suppose that print is a print queue manipulation interface used to manipulate queues for several types of printers. One printer type is **laser**, and objects of this type are **my_laser**, **your_laser**, and **public_laser**. The **print** interface includes the **print$add_to_queue**, **print$delete_from_queue**, and **print$check_queue** operations.

A remote matrix arithmetic package is another example of an interface. An array processor exports a set of matrix operations as an interface. The array processor is the object, and its type is **arrayproc**. Array processor objects are accessed through a **vector_ops** interface with operations such as **vector_add** and **vector_multiply**. The **arrayproc** type might have other interfaces, for example, a **misc_ops** interface, with operations such as **max_absolute_value** and **root_mean_square**, and a **scalar_ops** interface for scalar arithmetic. Client programs on various hosts import the **vector_ops** interface by making calls such as **vector_add**. The programs run on the local hosts, but all matrix operations run on the remote array processor.

Clients, Servers, and Managers

An RPC client is a program that makes remote procedure calls to request operations. A client does not know how an interface is implemented. The client is not required to know the location of the server exporting the interface either.

An RPC server is a process that implements the operations in one or more interfaces. The server is the module to which the RPC runtime library sends an operation request packet, and from which this library receives a response containing the results of the operation.

A server can export a single interface or multiple interfaces as explained in the array processor example discussed previously in Interfaces on page 5–7. The following figure illustrates a server that exports two interfaces. A server can export an interface for a single object or for multiple objects. In the array processor example, there is only one object, the array processor. A file server, however, manages many file objects.

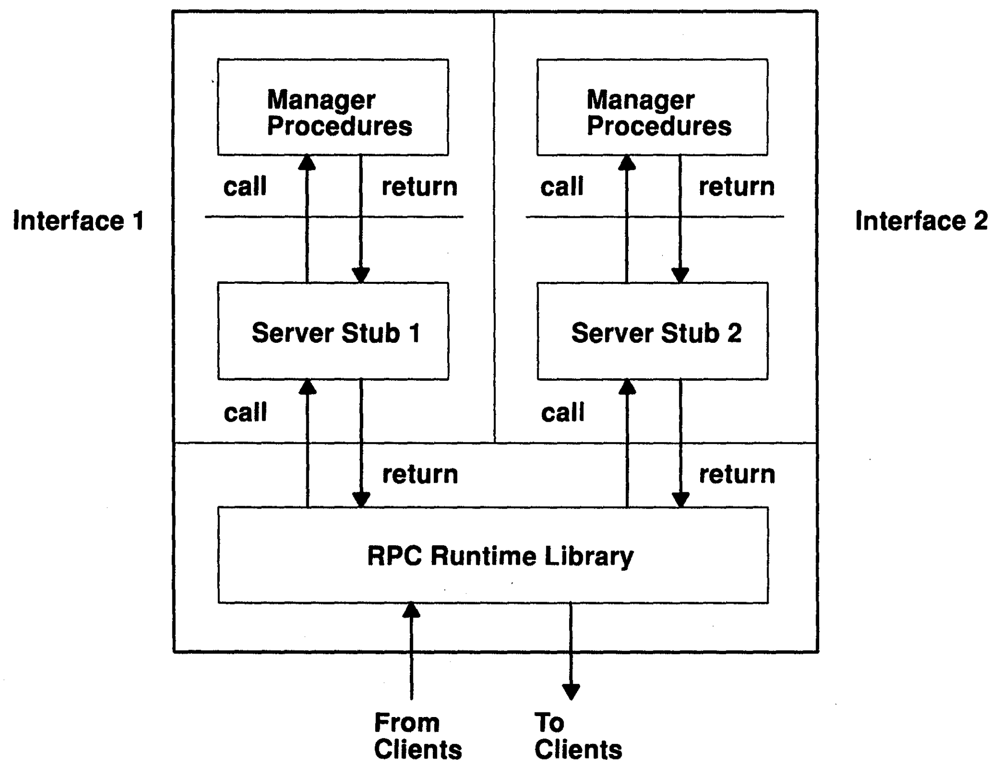


Figure 6. RPC Server Exporting Two Interfaces

A server can also be a client. It can even be a client of itself. The client and server play symmetrical roles in the RPC paradigm. However, their program structures are asymmetrical.

The client consists of two parts:

- The application code (labeled client), which makes calls to be executed remotely
- The stub code (labeled client stub), which uses the RPC runtime library to have these calls executed.

The server has the following parts:

- The manager code, which corresponds to the client application code
- A stub, which corresponds to the client stub
- The code that initializes the server process itself.

Reference to the server means the whole server process. However, manager code refers directly to the procedures that actually implement the server operations.

Identifying Objects and Servers

When a client makes a remote procedure call to request that a particular operation be performed on a particular object, the following information is required to transmit the call from the RPC runtime library:

- The object on which the operation is to be performed
- The server that exports the interface containing the operation.

This information about the object and the server is represented in the client process by a handle. Handles are created and managed by RPC library routines. Once a handle is created, it always represents the same object. The handle can, however, represent different servers at different times, although it is not required to specify a server at all. The representation of the server in a handle is called the binding. To bind a handle is to establish its representation of the server.

RPC Handles

An RPC handle is a pointer to an opaque data structure that includes the information required to access a remote object. Clients and servers do not manipulate this structure directly, but through RPC runtime library calls. The following figure shows an RPC handle.

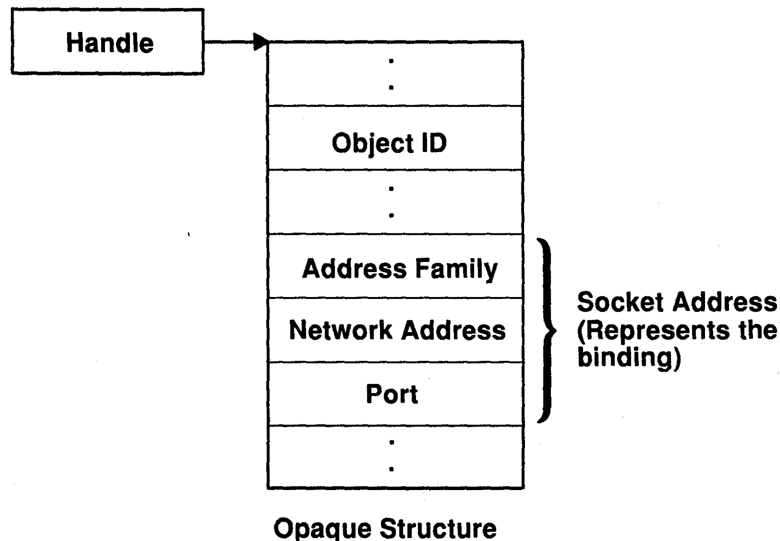


Figure 7. RPC Handle

The RPC Binding

A binding, and therefore the RPC handle that includes the binding, can exist in three states:

- Unbound (or allocated)
- Bound-to-host
- Bound-to-server (or fully bound).

An unbound or allocated handle identifies an object but does not identify its location. When a client uses an unbound handle to make a remote procedure call, the runtime library broadcasts a message to all hosts on the local network. Any host that supports the requested interface to the object (such as the interface that contains the called operation) can respond. The client runtime library accepts the first response that it receives.

A bound-to-host handle identifies the object and the host but does not represent the specific server that exports the interface to the object. When a client uses a bound-to-host handle to make a remote procedure call, the runtime library sends a message to the forwarding port on the specified host. If a server that exports the required interface to the object is registered with the host's Local Location Broker, the message is forwarded to the required server.

A bound-to-server or fully bound handle identifies the object and the server. When a client uses a fully bound handle to make a remote procedure call, the runtime library sends the message directly to the socket address identified by the handle.

In all cases, whenever the client RPC runtime library receives a response from a server, it binds the handle to the server socket address. Therefore, RPC handles are fully bound whenever a remote procedure call returns, and the client does not need to use the broadcasting or forwarding mechanism for subsequent calls to the server.

The following table shows, for each possible binding state of a handle when a remote procedure call is made, the information that the handle represents, the delivery mechanism of the remote procedure call, and the binding state when the procedure call returns.

Handles and Binding States

Binding State on Call	Information	Delivery	Binding State on Return
Allocated	Object	Broadcast to all hosts	Fully bound
Bound-to-host	Object Host	Sent to host's Location Broker forwarding port	Fully bound
Fully bound	Object Host Server	Sent to specific server port	Fully bound

Stubs

Both clients and servers are linked (in the sense of combining object modules to form executable files) with stubs. Stubs enable the clients and servers to use the RPC facilities as transparently as possible, which makes remote invocations look almost local. The client stub stands in for the remote procedures in the client process. The server stub stands in for the client in the server process. This means that, when a client makes a remote procedure call, it actually calls a routine in the client stub. The client stub calls an RPC runtime routine to send the request to the server. Similarly, the server RPC runtime library calls the server stub when it receives an RPC packet, and the server stub then calls manager code that executes the requested procedure.

The stub program modules transfer remote procedure calls and responses between an RPC client and the manager procedures that implement an interface. The modules convert data between the procedure call format specified by the interface definition and the format required by the RPC runtime routines. The modules also issue the RPC runtime library calls required for communication between the client and the server.

When a client calls an interface operation, such as **vector_add** from the array processor example, it actually calls a routine in the client stub. The client stub does the following:

1. Establishes the binding between the client and the server if the client has not explicitly created a binding.
2. Marshalls, or copies into an RPC packet, the input parameter values.
3. Calls an RPC runtime procedure to send the packet to the server stub and await a reply.
4. Receives the reply packet.
5. Unmarshalls the output parameter values into the format expected by the client. The format is specified in the interface definition.
6. Converts the output's data representation into a form that is meaningful to the client if the server uses a different format (for example, converts characters from EBCDIC to ASCII).
7. Returns to the client.

Similarly, the RPC runtime library calls a server stub routine when the server receives a request from the client. The server stub then does the following:

1. Unmarshalls the input parameter values into the format expected by the server. The format is specified in the interface definition.
2. Converts the input's data representation into a form that is meaningful to the server if the client uses a different format (for example, converts characters from ASCII to EBCDIC).
3. Calls the manager procedure that implements the operation.
4. Marshalls the output parameter values into an RPC packet.
5. Returns the packet to the RPC runtime library for transmission to the client stub.

The following figure illustrates these operations in the context of the figure illustrating a remote procedure call flow and the figure illustrating a server exporting two interfaces.

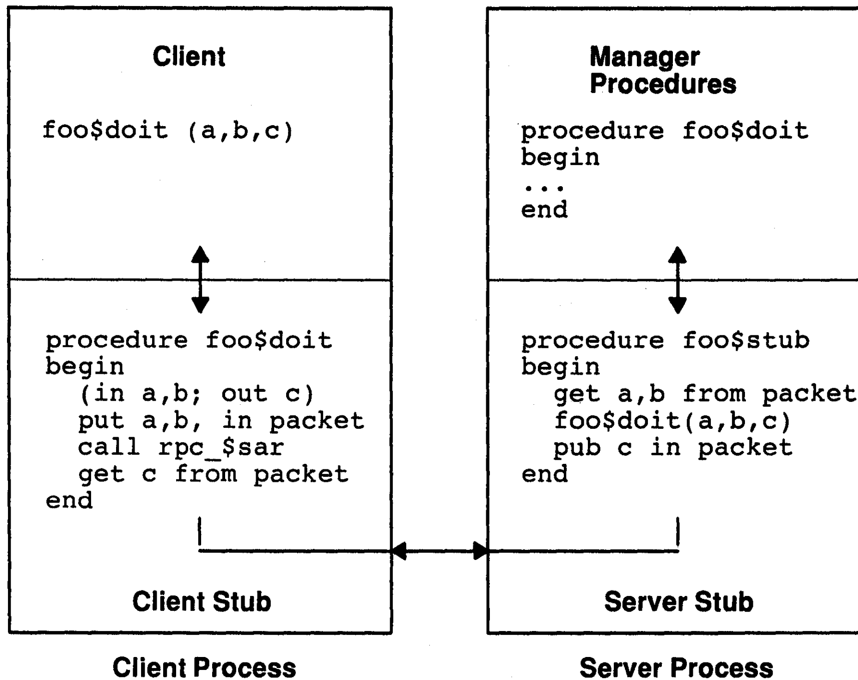


Figure 8. Client and Server Stub Operations

NCS provides a compiler that automatically generates source code for both the client and the server stubs from a definition of the interface written in Network Interface Definition Language (NIDL). Interface Definitions and the NIDL Compiler on page 5–16 provides more detailed information about the NIDL compiler and the stubs that it generates.

The Remote Procedure Call (RPC) Runtime Library (NCS)

The RPC runtime library, included in the **/usr/lib/libnck.a** library, contains the routines, tables, and data that support the communication of remote procedure calls between clients and servers. RPC runtime routines are responsible for transmitting RPC packets between the client and server stubs.

Routines

The RPC runtime library contains routines that are normally used only by clients (client routines), some that are normally used only by servers (server routines), and others that either clients or servers can use (conversion routines).

Client Routines

rpc_$alloc_handle
: Allocates a handle that identifies a specific object but not a specific server.

rpc_$set_binding
: Sets the binding in an allocated handle so that it specifies a socket address.

rpc_$bind
: Allocates a handle and sets a binding. This function is identical to a call to the **rpc_$alloc_handle** routine followed by a call to the **rpc_$set_binding** routine.

rpc_$dup_handle
: Makes a copy of an RPC handle.

rpc_$clear_server_binding
: Removes the association between an RPC handle and a server, but retains the association with a host. A remote procedure call that is made using this handle is sent to the Local Location Broker forwarding agent port on the remote host.

rpc_$clear_binding
: Removes the association between an RPC handle and both a server and a host. This routine saves the handle and its related resources for reuse in accessing the same object, possibly at another server or host location. A remote procedure call that is made using a handle with a cleared binding is broadcast to the Local Location Broker forwarding agent port.

rpc_$free_handle
: Removes (frees) the information represented by a handle by clearing the association between the handle and an object and socket address, and then releasing the RPC handle.

rpc_$inq_binding
: Returns the socket address identified by the RPC handle, which enables the client to determine the specific server that responded to a remote procedure call.

rpc_$sar
: Sends a packet to a bound interface and awaits a reply from the server.

Server Routines

rpc_$use_family
Creates a socket that is the server's end point for communications with clients over the network and assigns an available port number for the socket.

rpc_$use_family_wk
Creates a socket that uses a well-known port.

rpc_$register Registers an interface with the RPC runtime library, which enables the server to handle requests for the registered interface.

rpc_$unregister
Unregisters an interface that was previously registered with the server (with the **rpc_$register** routine). The server does not respond to requests for the unregistered interface.

rpc_$listen Listens for RPC requests from clients, calls the requested interface procedure when a request is received, and sends the result in a reply to the client.

rpc_$inq_object
Returns the UUID of the object represented by an RPC handle.

Conversion Routines

rpc_$name_to_sockaddr
Determines the socket address for a specific named host.

rpc_$sockaddr_to_name
Given a socket address, returns the host name and socket port number.

Client Routines

The RPC runtime routines that are called by clients include routines that either create handles or manage their binding state. In addition, there is one routine that sends and receives packets.

The client and its stub use handles to represent the object and server to the RPC runtime routines. Manual binding occurs when the client makes RPC handle management calls directly. Automatic binding occurs when the client stub calls a routine (written by the application developer) that makes all of the client's calls to the RPC runtime routines.

An internal call generated by the NIDL compiler, the **rpc_$sar** call (RPC Send and Await Reply), sends data and a request for a specific operation, and then awaits a reply from the server. This call is made only by the client stub. When the stub receives a response, the stub then passes the results to the calling application. Since the NIDL compiler automatically generates all **rpc_$sar** calls, clients never call this routine directly.

Server Routines

The RPC runtime routines that are called by servers initialize the server, except for one routine that identifies the object to which a client has requested access.

Most of the server routines in the RPC runtime library initialize the server so that it can respond to client requests for one or more interfaces. In the server code, routines should be included to do the following:

- Create one or more sockets to which clients can send messages.
- Register each interface that the server exports.
- Begin listening for client requests.

The RPC runtime library provides two routines that create sockets. One creates a socket with a well-known port while the other creates a socket with an opaque port number.

A single server can support several interfaces. It can also listen on several sockets at a time. Most servers use one socket for each address family. A server is not required to use different sockets for different interfaces.

The server must register each interface that it exports with the RPC runtime library so that the runtime library can direct client calls to the procedures that implement the requested operations. The library also includes a routine to unregister an interface that the server no longer exports.

Once the server creates sockets, registers its interfaces, and begins listening, it is not required to make additional calls to the initialization routines. However, a server can register and unregister interfaces while it is running.

The **rpc_$inq_object** routine enables the server's manager procedures to determine the specific object that the manager procedures must access. This routine returns the object UUID identified by an RPC handle. The **rpc_$inq_object** routine is required because a server can export an interface to multiple objects and because the handle passed as a parameter in remote procedure calls is an opaque object.

Conversion Routines

The RPC runtime library also provides two routines that convert between names and socket addresses. These routines enable programs to use names rather than addresses to identify server hosts. A client can accept a host name as input and then use the **rpc_$name_to_sockaddr** routine to convert the information for use in a call to an RPC binding routine. A server can use the **rpc_$sockaddr_to_name** routine to log the identity of a client.

Interface Definitions and the NIDL Compiler (NCS)

The Network Interface Definition Language (NIDL) is used to define remote interfaces. NIDL definitions are used as specifications for application writers and as input to the NCS NIDL compiler.

An interface definition written in NIDL completely defines the interface and fully specifies each remote procedure call's parameters. This definition provides the information you need to develop clients that use the interface's operations.

NIDL contains constructs for describing the data types, functions, and procedures associated with a remote interface. It is a strictly declarative language and contains no executable constructs. NIDL can be written in syntax for C or Pascal language programmers.

The NIDL compiler translates an NIDL interface definition into C source-code stubs, which are then compiled and linked with clients and servers. These stubs facilitate remote procedure calls by copying arguments to and from RPC packets, converting data representations as necessary, and calling the RPC runtime library. It is much easier to write an NIDL interface definition than it is to write the code that the NIDL compiler generates for you. The compiler also generates C and Pascal header files. The following figure illustrates the NIDL compiler input and output files:

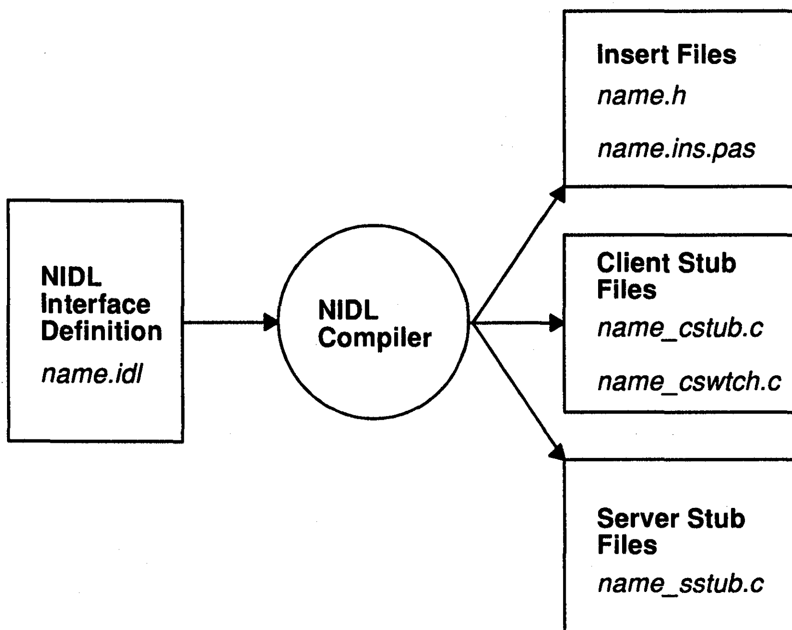


Figure 9. NIDL Compiler Input and Output Files

The NIDL compiler generates C source-code stub and switch files that are fully compatible with Pascal programs. If you write your client or server in Pascal, the stub and switch files can be compiled with the C compiler and linked to the Pascal object code. The stub and switch files can also be linked with FORTRAN programs.

Note: To compile NCS applications that use AIX Version 3.1, the applications must be linked with the **libbsd.a** library because of the usage of Berkeley Software Distribution (BSD) signals.

The Banking Example

The examples used here are based on a simple banking program. This banking program allows tellers to access and modify account databases at several banks. It mimics the real world, where you can use a single automatic teller to access accounts at any of several different banks. The Banking Example figure illustrates this banking example, showing one possible implementation using the banking client and server programs. In this figure, clients (teller programs) run on hosts A and B. The clients can access accounts belonging to either of two different banks. One bank's account database is maintained by a server running on host C. A second bank's accounts are maintained on host D.

This example uses remote procedure calls to access one type of object, the bank database, which maintains the information for all of the bank's accounts. Each bank database object contains a number of entries, each with the following information:

- Account owner's name
- Account balance
- Account number
- A Boolean deleted marker
- Time of last transaction
- Time of account creation.

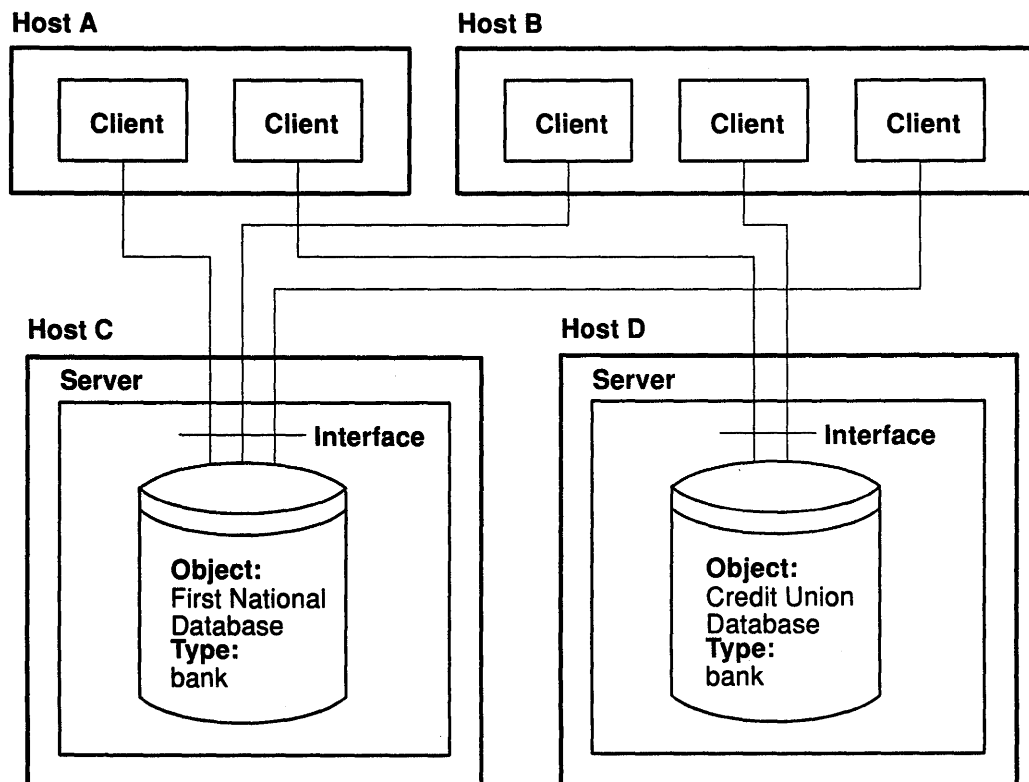


Figure 10. The Banking Example

There is one copy of the database object for each bank. Multiple bank database objects are allowed.

An interface called bank contains all operations that are supported for bank database objects. The following describes these operations:

bank$create_acct Given a customer name, creates a new account and returns the account number.

bank$inq_acct Given an account number, returns the current balance, time of last transaction, and time the account was created.

bank$get_acct Returns the account number that corresponds to a specific customer name.

bank$kill_acct Cancels the account with the specified account number.

bank$deposit Updates the balance of the specified account to reflect a deposit.

bank$withdraw Updates the balance of the specified account to reflect a withdrawal.

The **/usr/lpp/ncs/examples/bank/bank.idl** file contains the NIDL definitions for these operations.

The binop Example

The following example programs show the interface definition for the binop application that is provided in the **/usr/lpp/ncs/examples** directory. The **/binop_c/binop.idl** file, written in C syntax, follows:

```
%c
[uuid(334a2e240000.0d.00.00.31.66.00.00.00), port(dds:[19],
ip:[6677]),
                                                 version(0)]
interface binop
{
import '/usr/include/idl/base.idl';
import '/usr/include/idl/rpc.idl';

void [idempotent] binop$add(
           handle_t [in]  h,
           int      [in]  a,
           int      [in]  b,
           int      [out] *c
           );
}
```

The **binop/binop.idl** file, written in Pascal syntax, follows:

```
%pascal
[uuid(334a2e240000.0d.00.00.31.66.00.00.00), port(dds:[19],
ip:[6677]),
                                                   version(0)]
interface binop;

import
        '/usr/include/idl/base.idl',
        '/usr/include/idl/rpc.idl';

[idempotent] procedure binop$add(
            in  h: handle_t;
            in  a: integer32;
            in  b: integer32;
            out c: integer32
            );
end;
```

The first line of each interface definition states the syntax of the particular NIDL being used.The next three lines of the interface definition identify the interface. These lines specify the interface's unique identifier, well-known ports, version, and name. The interface UUID is the name used by the RPC runtime library and by the Location Broker. The textual name **binop** is used for documentation and for generating names in the stub and insert files.

The definition then imports two existing interfaces. The NIDL **import** statement is similar to the C **#include** or the Pascal **%include** statement, with some differences. (The differences are discussed in Using C Syntax with NIDL on page 5–50 and Using Pascal Syntax with NIDL on page 5–62.)

The remaining lines define a remote procedure. The first argument contains RPC binding information, which the RPC runtime library uses to send the call to the correct server. The next two arguments are inputs. The final argument is an output.

To keep this example simple, well-known ports and predefined data types are used in the interface definition.

Running this definition through the NIDL compiler produces the following output files:

- The header file (**binop.h**)
- The client stub and switch files (**binop_cstub.c** and **binop_cswtch.c**)
- The server stub file (**binop_sstub.c**).

Note: If the application is written in the Pascal language, you must use the **–pascal** flag with the NIDL compiler. This produces another header file, **binop.ins.pas**, which contains Pascal **constant**, **type**, and **procedure** declarations.

The **binop.h** file contains C **#define** control lines, **typedef** declarations, and **function** declarations. In addition, the **binop.h** file and, if you have produced it, the Pascal header file contain the interface specification, a data structure that is passed to the RPC runtime library when the interface is bound or registered.

The **binop_cstub.c** and **binop_cswtch.c** files together implement the client stub. The files contain the **binop$add** procedure. This procedure copies its two input arguments (`a` and `b`) into an RPC packet and calls the **rpc_$sar** (send and await reply) routine. When **rpc_$sar** returns, the output argument is copied from the packet into the output argument (`c`) of the procedure, and then the procedure returns to its caller.

The **binop_sstub.c** contains the server stub for the binop interface. The procedure takes a packet as its input argument. It copies the *a* and *b* values from the packet into two local variables. It then calls the **binop_$add** manager procedure with the values. The result of calling this procedure c is copied into the packet, and control is returned to the RPC runtime library. The RPC runtime library sends the packet back to the waiting client, which processes it as described above.

Stub Functionality

Stubs are used to convert and copy data, or to bind with a remote interface. The simplest stubs provide only argument passing and data conversion. More complex stubs aid in the RPC binding process.

Marshalling and Conversion

All stubs marshall values into RPC packets and unmarshall values from RPC packets. Client stubs marshall input parameters, call the **rpc_$sar** routine to send the packet to the server and await a reply, and unmarshall the output parameters. Server stubs unmarshall input parameters from a received packet, call the interface manager to process the data, and then marshall the output parameters to be returned to the client.

The compiler automatically generates stubs that can marshall and unmarshall the following data types:

- Signed and unsigned integers
- Single-precision and double-precision floating-point numbers
- Characters
- Strings
- Fixed-length and variable-length arrays
- Enumerations
- Sets
- Records
- Discriminated unions
- Simple pointers.

The compiler does not generate code to marshall and unmarshall either pointers to pointers or records that contain pointers. However, with NIDL, you can write separate procedures that handle such types, which means data structures of almost arbitrary complexity can be passed between machines.

In addition to marshalling the data, each stub procedure checks an incoming packet's data representation format. Each side sends data using its native format, and the transmitted packet indicates the sender's data representation for integers, characters, and floating-point numbers. If the sender's representation of a data type is different from the receiver's representation, the receiving stub converts that data type when it unmarshalls values. If the sender and receiver have the same representation for a scalar type, conversion is not performed.

Note: This technique, called *receiver makes it right*, the sender never puts data into a canonical form. There is no need to convert data to a standard form if both machines have identical representations. This strategy allows heterogeneity at minimum cost.

Handles and Binding

The RPC runtime library (specifically, the **rpc_$sar** operation that the NIDL compiler includes in the stub) must know the object and binding represented by an RPC handle before it can send a remote procedure call. NIDL supports the following techniques for managing this information:

- Explicit or implicit interface handles, which determine whether the client uses operation parameters or a global variable to represent the handle information
- Manual or automatic binding, which determines whether the client or the stub generates the actual RPC handle that the client runtime routine uses to send a request.

The effect of the interface-defined handle format and binding technique on the handle variable data type and format can be summarized as follows:

Explicit and Implicit Handle Bindings		
Type of Handle	**Manual Binding**	**Automatic Binding**
Explicit	Data type: **handle_t**	Data type: user defined
Explicit	Format: operation parameter	Format: operation parameter
Implicit	Data type: **handle_t**	Data type: user defined
Implicit	Format: client global variable	Format: client global variable

The following sections describe handles and binding in detail.

Explicit and Implicit Handles

In an RPC application that uses explicit handles, each operation in the interface must have a handle parameter. The client explicitly passes the handle parameter whenever it makes a remote procedure call. Similarly, this parameter is passed to the server's manager routines when the server receives a remote procedure call.

In an application that uses implicit handles, the handle identifier is a single global client variable. The operations do not have to include a handle parameter, and the server does not receive the handle information. This means an implicit handle makes remote procedure calls look more like ordinary procedure calls, since there is no need to pass special information in each call. An implicit handle trades the flexibility of an explicitly passed parameter for the simplicity of a single global variable.

Implicit handles imply idempotent procedures. That is, if both client and server are on two distinct networks and the client broadcasts on both networks, the server receives both broadcasts and executes the procedure one time for each request. The two requests are different because the socket address (sockaddr) of the sending client is different.

The following figure illustrates explicit and implicit handles:

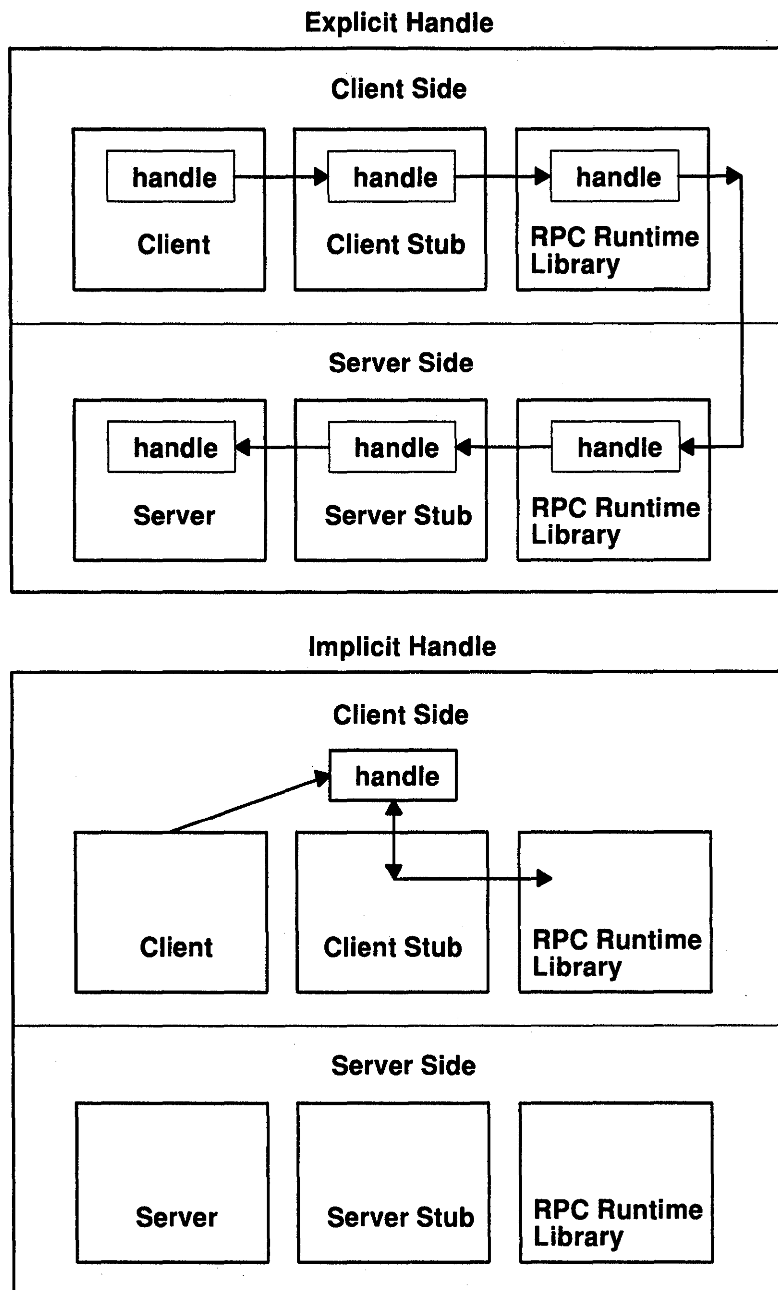


Figure 11. Explicit and Implicit Handles

An implicit handle can be useful if the client accesses only one server while it is running and if the server manages only one object. An array processor application can meet these criteria. The client starts by locating a particular array processor and uses the server at that host until it completes its calculations. The array processor server manages only the array processor object.

Interfaces using an implicit handle have two major limitations that interfaces using explicit handles do not have. First, the handle is not passed to the server routines, and the server therefore does not receive the object identifier that is represented by the handle. As a result, you cannot use implicit handles in any interface whose server manipulates multiple objects unless you explicitly pass an object identifier (such as a UUID or a path name) as an operation parameter.

A second limitation of using implicit handles is that, because binding information is contained in one variable, the client cannot access more than one remote server at any time. For example, you cannot use an implicit handle for an application that divides computation among multiple processors because the single handle restricts you to using only one remote processing server at a time.

Manual and Automatic Binding

In an RPC application that uses manual binding, the client makes all calls that create and manage the handle. As a result, the interface's handle variable must be an RPC handle, and the client makes all the RPC runtime library calls that allocate, bind, and manage this handle.

In an application that uses automatic binding, the client does not manage the binding. Instead, the client stub calls an autobinding routine each time the client makes a remote procedure call and calls an auto-unbinding routine after the remote call returns. The type of the handle variable can be application dependent. However, the variable must provide information sufficient for the binding routine to generate an RPC handle. The binding and unbinding routines must be written by the application developer.

Automatic binding is useful when a procedure talks to different servers each time it is invoked. A good example is an interface to a remote file system. On any open call, you can be operating on a file located at a different host. It is unreasonable to expect a client of the file system to call an RPC binding routine before each file access. Therefore, the file interface should use automatic binding. This means that the client passes the file path name as its handle variable in each procedure call. The stub then passes the path name to the autobinding routine, which generates an RPC handle. The stub can then use the RPC handle to send the request to the client.

Automatic binding trades performance for convenience. Each time it processes a remote procedure call, the stub must convert between a handle that is meaningful to the client and an RPC handle that specifies an object UUID and socket address. Interfaces that use automatic binding require more processing than those in which the client does the binding once and passes an RPC handle (explicitly or implicitly) to the stub.

The following table shows the differences between manual and automatic binding in making a remote procedure call.

Manual and Automatic Binding in a Remote Procedure Call	
Manual Binding	**Automatic Binding**
Client: Generates an RPC handle. Binds handle, if necessary. Makes procedure call to stub.	Client: Using handle, makes procedure call to stub.
Stub: Sends request to server. Receives response from server. Returns to client.	Stub: Calls autobinding routine.
Client: Receives call return from stub. Manages handle, if necessary.	Autobinding routine: Generates and binds RPC handle, as necessary, based on passed handle. Returns RPC handle to stub.
	Stub: Sends request to server. Receives response from server. Calls auto-unbinding routine.
	Auto-unbinding routine: Clears or frees handle, as necessary. Returns to stub.
	Stub: Returns to client.

Client Switches

Although the RPC paradigm calls for a single client stub, the NIDL compiler actually generates two client files: a stub (*name*_**cstub.c**) and a switch (*name*_**cswtch.c**).

Writing Programs That Use the Network Computing System

The following sections contain information on writing applications that use the Network Computing System. It is important to understand how NCS uses UUIDs, how it defines the interface definitions, and how to manage handles and bindings.

Managing UUIDs

Each object, type, and interface must have a UUID. You must generate a new UUID each time you create an object, type, or interface. The following tools are provided for managing UUIDs:

- The **/etc/ncs/uuid_gen** command, which generates character-string representations of UUIDs
- Three **uuid_$** library routines for generating UUIDs and for converting between UUIDs and their character-string representations.

The uuid_gen Command

The **uuid_gen** command returns a character-string representation of a UUID. You should use this program to generate the UUID specifier that you include in each interface description heading. You can also use the **uuid_gen** command to generate object-type

UUIDs that you administer manually. The following example shows the **uuid_gen command** run from a shell:

```
/etc/ncs/uuid_gen
```

This produces the following output:

```
33547f280000.0d.00.00.37.27.00.00.00
```

For a detailed explanation, refer to the **uuid_gen** command.

UUID Library Routine Summary

The **uuid_$** library routines are summarized as follows:

uuid_$gen Generates a new UUID.

uuid_$decode Converts a character-string representation of a UUID (as generated by the **uuid_gen** command) into a **uuid_$t** value (that can be used by a program).

uuid_$encode Converts a UUID into its character-string representation.

Refer to uuid_$ Library Routines on page 5–94 and the individual library routines for more detailed information.

Defining the Interface

An interface is defined by writing an interface definition in NIDL. This interface definition specifies the signatures of the remote procedure calls (operations) that the clients call and the servers implement. It also provides other information that the NIDL compiler uses to generate the client and server stubs. The **/usr/lpp/ncs/examples/bank** file contains the NIDL definition for the bank example interface using Pascal syntax.

An interface definition consists of the following:

- Heading
- Import declarations
- Constant declarations
- Type declarations
- Operation declarations.

The Heading

The heading of an interface definition specifies whether the definition is written in C or Pascal language syntax, and defines the interface name and attributes. The following attributes can be specified:

uuid The Universal Unique Identifier assigned to this interface. No other object, type, or interface can be assigned this UUID.

version The version number of the interface. The RPC runtime code checks that the server and client both use the same version.

implicit_handle
The global variable containing handle information. If this attribute is not specified, the handle must be passed as an explicit parameter to each operation.

port The well-known port or ports on which the server exporting this interface listens.

local A flag specifying that this interface definition is used only to generate header files (**.h** or **.ins** files) and contains information only about constants and data types, not procedures. The compiler does not create any stub files.

The heading must specify the interface name and either the **uuid** or the **local** attribute. All other attributes are optional. Their use depends on the interface you are creating. The following is an example of the NIDL heading in Pascal syntax:

```
%pascal
[ uuid(334033030000.0d.00.00.87.84.00.00.00), version(0) ]
interface bank;
```

The banking example on page 5–17 uses explicit handles and manual binding. Therefore, each procedure must have a **handle_t** handle variable as its first parameter, as shown in the following NIDL declaration in Pascal syntax of the **bank$kill_acct** call:

```
procedure bank$kill_acct(
    in h:      handle_t;
    in acct:   bank_$acct_t;
    out st:    status_$t
    );
```

If the banking example used implicit handles and manual binding, the NIDL heading would be as follows:

```
%pascal
[ uuid(334033030000.0d.00.00.87.84.00.00.00), version(0),
  implicit_handle(bank_handle: handle_t)]
interface bank;
```

The NIDL declaration of the **bank$kill_acct** operation is as follows:

```
procedure bank$kill_acct(
    in acct:        bank$acct_t;
    out st:         status_$t
    );
```

Import, Type, and Constant Declarations

NIDL provides declarations for included files, data types, and constants similar to those in the C and Pascal languages.

The NIDL **import** declaration is similar to the C **#include** and Pascal **%include** directives. It specifies another NIDL file whose definitions are to be used by the importing interface. For example, if an interface includes operations that use **rpc_$** routines or types, the definition for that interface must import the **rpc.idl** file.

NIDL provides three type attributes that you can specify for certain data types. Two of these attributes, **last_is** and **max_is**, are used with array data types. These attributes control the amount of data transmitted between the client and server and the amount of storage allocated at the server.

The **transmit_as** attribute allows you to make remote procedure calls that use complex data types such as trees and linked lists. The NIDL compiler cannot generate code to marshall and unmarshall data types that include pointers (that is, pointers to pointers and records that contain pointers). However, NIDL allows you to write routines that convert these types into transmissible types. The **transmit_as** attribute specifies the form in which the complex data type will be transmitted. It indicates to the NIDL compiler that user-written routines should be called to do type conversion and storage management.

Operation Declarations

Operation declarations specify the format of each remote procedure call, including the procedure name, the type of data returned (if any), and the types of all parameters passed in the call. Operation declarations also specify parameter and operation attributes that the NIDL compiler uses in generating stubs. The following example illustrates, first in C syntax and then in Pascal syntax, the operation declaration for the **bank$get_acct** procedure:

```
%c
...
[idempotent] void bank$get_acct(
    handle_t [in] h,
    bank$acct_name_t [in] name,
    long [in] namelen,
    bank$acct_t [out] *acct,
    status_$t [out] *st
    );

%pascal
...
[idempotent] procedure bank$get_acct(
    in h:         handle_t;
    in name:      bank$acct_name_t;
    in namelen:   integer32;
    out acct:     bank$acct_t;
    out st:       status_$t
    );
```

Operation Attributes

Operation attributes are the first part of an operation declaration and are used to describe characteristics of individual operations that affect communication between server and client. You can specify any of the following operation attributes and attribute combinations:

- **idempotent**
- **broadcast**
- **maybe**
- **broadcast, maybe**

An operation is idempotent if its results do not affect the results of any operation, including itself. For example, a call that returns the time is idempotent because, while the operation returns different results each time it is called, the call itself has no effect on any operations. If you specify the idempotent operation attribute in an operation declaration, the RPC runtime library takes advantage of the operation's idempotent characteristic to reduce internal overhead and thereby speed up the calling activity. The **bank$get_acct** operation is idempotent.

The **broadcast** attribute is useful for applications in which many servers should be notified of an event. The **maybe** attribute is useful when there is no need for confirmation that a message was received. For example, the Location Broker Client Agent uses both the **broadcast** and **maybe** attributes for routines that update the GLB database.

A distributed game can use these attributes for routines that inform all players of each move, as in the following example:

```
%c
...
[broadcast, maybe] void dgame$my_move(
    char [in] my_name[64],
    move_record_t [in] my_move
    );

%pascal
...
[broadcast, maybe] procedure dgame$my_move(
    in my_name: array [1..64] of char;
    in my_move: move_record_t
    );
```

Note: This operation declaration does not specify a handle parameter; the interface uses an implicit handle.

Handles and Binding

An implicit handle is specified in the heading as an interface attribute. If an interface uses explicit handles, however, the handle must be supplied as the first parameter in each operation declaration. The type of a handle, whether implicit or explicit, determines how the application manages the binding.

If the operation declaration (or, for implicit handles, the interface heading) specifies the type as **handle_t**, then the interface is manually bound. This means that the client must manage the binding, and pass to the stub a handle of type **handle_t** that is meaningful to the underlying RPC runtime routines.

If the operation declaration (or the interface heading) specifies any other type for the handle parameter, then the interface is automatically bound. In this case, whenever the stub receives a call for the operation, it calls a binding routine named *type*_**bind**, where *type* is the data type specified for the handle parameter. The stub then uses the **handle_t** variable returned by *type*_**bind** to send the remote procedure call. When the RPC runtime library returns a response, the stub automatically calls a *type*_**unbind** procedure, and then returns the response to the client. The application developer must write the *type*_**bind** and *type*_**unbind** routines that convert between the handles that are meaningful to the client and the RPC handles required by the runtime library.

Directional Parameter Attributes and Parameter Classes

The directional characteristics of an operation parameter are specified by the parameter attribute in the C syntax of NIDL and by the parameter class in Pascal syntax. Attributes and classes are different syntactic elements, but are expressed with the same key words. The parameter attributes and classes inform the NIDL compiler and the RPC runtime library whether a parameter passes from client to server (**in**), from server to client (**out**), or both.

You must specify exactly one of the following combinations for each parameter:

in Specifies that the client passes the parameter to the server.

in ref (Pascal syntax)
Specifies that the client passes the parameter to the server and that the parameter is passed by reference. Note that **in ref** is defined only as a parameter class in the Pascal syntax of NIDL. Passing by reference is specified in the C syntax of NIDL by the & operator. All Pascal parameters that are longer than four bytes, except for arrays, must have the **in ref** class.

out Specifies that the server passes the parameter to the client.

in, out (C syntax)

in out (Pascal syntax)
Specifies that the parameter is passed in both directions.

Attributes for Array Parameters

The following are optional attributes for array parameters:

last_is (*last*) Specifies that *last* is the index of the last array element to be passed between the client and the server. This attribute is required for open (that is, variable-length) arrays.

max_is (*max*) Specifies that *max* is the maximum possible index of an open array.

The following example illustrates, first in C syntax and then in Pascal syntax, the use of parameter attributes in an operation definition. In this example, the `data_record_t` is a record type declared elsewhere in the NIDL file.

```
%c
...
void update_record(
    handle_t [in] h,
    long [in] code,
    [last_is (name_end)] char [in] namelen[64],
    long [in] name_end,
    data_record_t [in] *data_rec,
    status_$t [out] *st
    );

%pascal
...
procedure update_record(
    in h:                           handle_t;
    in code:                        integer32;
    in name: [last_is (name_end)] array [1..64] of char;
    in name_end                     integer32;
    in ref data_rec:                data_record_t;
    out st:                         status_$t
    );
```

Running the NIDL Compiler

Once you have written the interface definition, use the following command to run the NIDL compiler:

```
nidl File [options...—cpp]
```

The *File* parameter specifies the path name of the interface definition file. When run without options, the compiler generates the C header file, the client stub and switch files, and the server stub file. For a detailed explanation of the *options* choices, see the **nidl** command.

Header Files and NIDL-Defined Variables

The NIDL compiler creates header files containing the following:

- File inclusion directives
- Constant declarations
- Type declarations
- Operation declarations
- Variable declarations and assignments.

In addition to code that is derived directly from analogous NIDL code (for example, a C **#include** directive derived from an NIDL **import** declaration), the header file contains code that defines the following types and variables:

interface$**if_spec**
: The interface specifier, required as an input parameter to some RPC routines. This variable identifies the interface and specifies such information as the interface version number, the well-known server port number (if any), the number of operations, and the interface UUID.

interface$**epv_t**
: The interface entry point vector (EPV) type, consisting of pointers to routines that correspond to the operations in the interface. This is the data type of the *interface*$**client_epv**, *interface*$**server_epv**, and *interface*$**manager_epv** variables.

interface$**client_epv**
: The EPV that is used to access client stub routines. The client switch uses this EPV. Replicated servers also use this EPV. The client stub assigns the values in this EPV.

interface$**server_epv**
: The EPV that is used to access server stub routines. The RPC runtime library uses this EPV. The server must specify this EPV in the call to the **rpc_$register** routine that registers the interface with the RPC runtime library. The server stub assigns the values in this EPV.

interface$**manager_epv**
: An EPV that can be used to access the manager routines that implement the interface operations. This EPV is currently unused.

Marshalling and Unmarshalling of Complex Types

The NIDL compiler cannot generate stub code to marshall and unmarshall data types that include pointers, such as trees and linked lists. To use complex types in remote procedure calls, routines must be constructed that convert the complex type into transmissible type. The transmissible type represents the complex type in RPC communications. Each complex type in the NIDL definition can be identified with the **transmit_as** attribute. When the application is built, linking the conversion routines written by the programmer with the client and the server stubs that are generated by the NIDL compiler causes the stubs to call the routines as required.

Converting Complex Types

For each complex type you use, you must write the following four conversion routines:

- *type*_**to_xmit_rep** (*complex*, *pointer-to-transmissible*)
- *type*_**from_xmit_rep** (*transmissible*, *pointer-to-complex)*
- *type*_**free** (*pointer-to-complex*)
- *type*_**free_xmit_rep** (*pointer-to-transmissible*).

In all of these signatures, *type* is the name of the complex type, *complex* is a variable of this type, and *transmissible* is a variable of the transmissible type.

The *type*_**to_xmit_rep** routine takes as an input parameter a variable of the complex type. It takes as an output parameter a pointer to a variable of the transmissible type. This routine allocates storage for the transmissible type and converts from the complex type to the transmissible one.

The *type*_**from_xmit_rep** routine takes as an input parameter a variable of the transmissible type. It takes as an output parameter a pointer to a variable of the complex type. This routine allocates storage for the complex type and converts from the transmissible type to the complex one.

The *type*_**free** routine frees storage used by the server for the complex type.

The *type*_**free_xmit_rep** routine frees storage used by the client for the transmissible type.

The following example uses a calendar program that schedules appointments involving several users. The names of the users are stored in a linked list of the **cal$user_list_t** type. A list of this type is transmitted as a structure of the **cal$user_array_t** type. The following shows the NIDL definition in C syntax for the interface:

```
%c
[
uuid(34fe9783f000.0d.00.00.12.b9.00.00.00),
version(1)
]
interface calendar {

    const int MAX_NAME_LEN = 128 ;

    typedef
        char cal$user_name_t[MAX_NAME_LEN] ;
```

```
    typedef
        struct {
            int             n_users ;
            [last_is(n_users)]
                            cal$user_name_t participants[*] ;
        } cal$user_array_t ;

  typedef [transmit_as(cal$user_array_t)]
        struct {
            char                name[MAX_NAME_LEN];
            cal$user_list_t     *next ;
        } cal$user_list_t ;

    cal$make_appt(handle_t          [in] h,
                  cal$user_name_t [in] withWhom,
                  cal$user_list_t [in] participants) ;

}
```

The routines that convert between lists of the **cal$user_list_t** and **cal$array_t** types are as follows:

```
#include "calendar.h"
#include <stdio.h>


/*
 *  Client side call to convert a user list into its transmissible f
 * form.
 */


void cal$user_list_t_to_xmit_rep(pl, pa)
    cal$user_list_t  *pl ;
    cal$user_array_t **pa ;
{
    cal$user_array_t *ap ;
    int               pc ;
    cal$user_list_t   *up ;


    /*
     *  Count up the number of participants
     */


    pc = 1;
    for (up=pl;pl; pl=pl->next)
        ++pc ;


    /*
     *  Allocate the participant array and copy the participants
    *  into it.
     */
```

```
    ap = (cal$user_array_t *)
malloc(sizeof(cal$user_array_t)+(pc-1)*sizeof(cal$user_name_t));
    ap->n_users = pc ;
    pc = 0 ;
    for (up=pl; up; up=up->next) {
        strcpy(ap->participants[pc], up->name) ;
        ++pc;
    }

    *pa = ap ;
}

/*
 *  Client side call to free the transmissible representation.
 */

void cal$user_list_t_free_xmit_rep(pa)
    cal$user_array_t *pa ;
{
    free(pa) ;
}

/*
 *  Server side call to convert a user array from its transmissible
 * form.
 */

void cal$user_list_t_from_xmit_rep(pa, pl)
    cal$user_array_t *pa;
    cal$user_list_t  **pl ;
{
    cal$user_list_t *participants ;
    cal$user_list_t *pp ;
    int             i ;

    participants = (cal$user_list_t *)
   malloc(sizeof(cal$user_list_t));
    participants->next = NULL ;
    strcpy(participants->name, pa->participants[0]) ;

    pp = participants ;
    for (i=1; i<pa->n_users;i++) {
        pp -> next = (cal$user_list_t *)
      malloc(sizeof(cal$user_list_t)) ;
        pp = pp->next ;
        pp -> next = NULL ;
        strcpy(pp->name, pa->participants[i]) ;
    }

    *pl = participants ;
}
```

```
/*
 *  Server side call to free the complex representation.
 */

void cal$user_list_t_free(pl)
    cal$user_list_t *pl ;
{
    free(pl) ;
}
```

Notes:

1. The **transmit_as** attribute enables you to use any complex data type in a remote procedure call, provided you assume responsibility for converting between the complex type and a transmissible one.
2. You cannot use a complex data type as an element of an array or as a field in a record. This means complex types must appear at top level. You also cannot name one complex type as the transmissible representation for another complex type.
3. To compile, use:

   ```
   cc -c cal.c -I/usr/include/ude/idl/c -B/etc/ncs/ -tp
   ```

Managing Handles and Bindings

An NCS client application must identify the remote object that it is trying to access. The object is identified to the RPC runtime library in the form of an RPC handle. The application can use either of the following techniques to create and manage the RPC handle:

- Manual binding, where the client creates and manages RPC handles directly.
- Automatic binding, where the client does not manage the RPC handles and instead uses some other type of variable (a generic handle) to represent the required object. Whenever the client makes a remote procedure call, the stub calls a user-written autobinding routine that converts the generic handle into an RPC handle.

RPC Binding Management Routine Overview

The RPC runtime library includes calls that create and manage the binding state of an RPC handle (that is, its representation of a server and a host). The following figure illustrates several of these calls and their effect on an RPC handle:

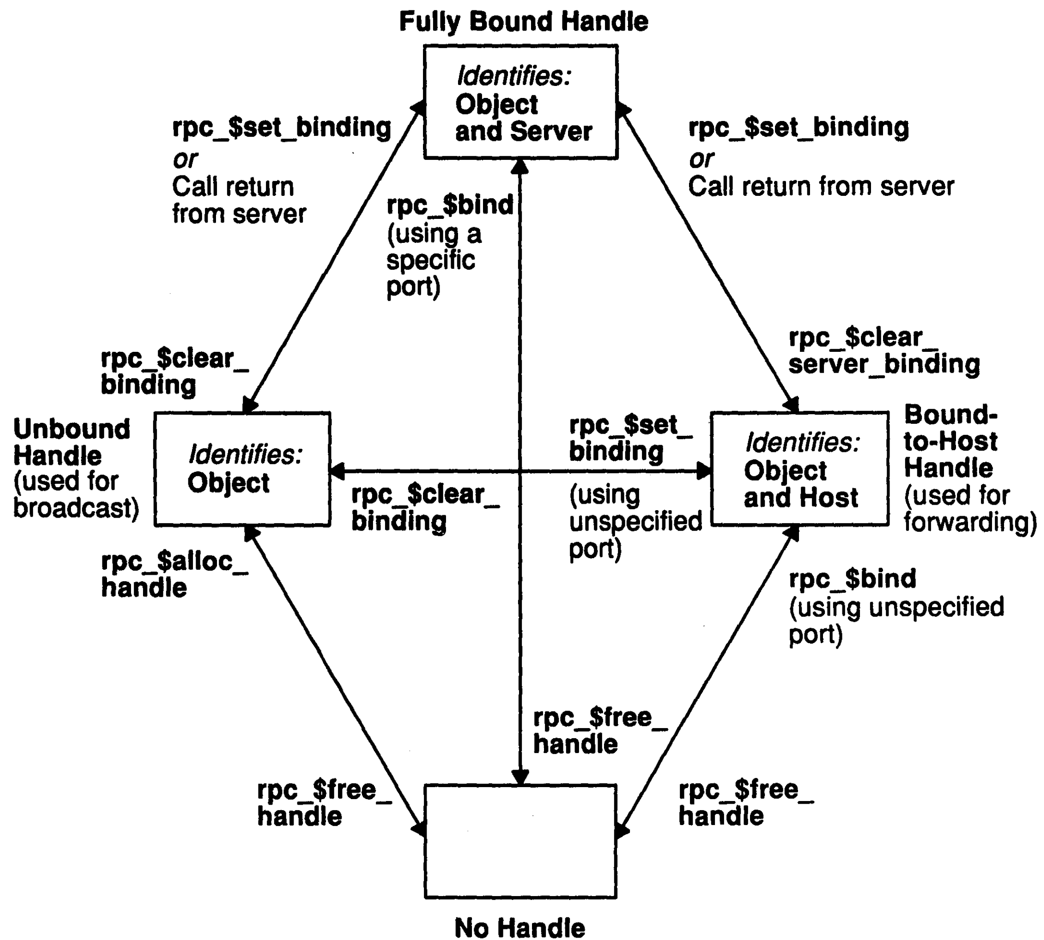


Figure 12. Managing RPC Handles

Obtaining a Socket Address

The socket address information represented in a handle determines the destination of any remote procedure call made using that handle. You can use an unbound handle to locate objects by a network broadcast, but if you want to specify a host, you must generate a socket address. NCS provides two tools for generating socket addresses: the Location Broker and the **rpc_$name_to_sockaddr** routine. The **rpc_$inq_binding** routine returns the socket address represented by an RPC handle.

Using Location Broker Lookup Routines

The Location Broker looks up and returns the socket addresses of servers. The Location Broker provides the following routines to handle lookup requests in its databases by object, type, interface, or any combination of these identifiers:

lb_$lookup_object
: Finds in the GLB database one or more entries that match the specified object identifier.

lb_$lookup_type
: Finds in the GLB database one or more entries that match the specified type identifier.

lb_$lookup_interface
: Finds in the GLB database one or more entries that match the specified interface identifier.

lb_$lookup_object_local
: Finds in the specified LLB database one or more entries that match the specified object identifier.

lb_$lookup_range
: Finds in the specified database (LLB or GLB) one or more entries that match the specified combination of object, type, and interface UUIDs.

See lb_$ Library Routines on page 5–89 and the individual library routines for more detailed information about the Location Broker library routines.

A lookup call for an object, type, or interface specifies the following:

object Specifies the UUID of the object being looked up.

object_type Specifies the UUID of the object type being looked up.

object_interface Specifies the UUID of the interface being looked up.

lookup_handle Specifies where to start in the database.

max_results Specifies the number of matching entries to return.

num_results Specifies the number of entries returned in the array.

results Specifies an array of matching entries from the call.

status Specifies the completion status.

For example, the bank server program, **bankd.c**, uses the following call to see if another bank server is already running:

```
lb_$lookup_object(&bank_id, NULL, MAX_LOCS, &n_locs, bank_loc, &st);
```

The NULL value for the *lookup_handle* parameter indicates that the lookup should begin at the start of the GLB database. The *MAX_LOCS* parameter specifies the maximum number of entries that the routine can return (there can be more). The *n_locs* parameter specifies the number of entries that are actually returned. The *bank_loc* parameter specifies an array of returned entries.

The *max_results* parameter, which cannot exceed the length of the *results* array, determines the maximum number of entries that a lookup routine can return. If a routine returns *max_results* entries before the entire database has been searched, the returned value of

lookup_handle is a key to the next unsearched part of the database. Otherwise, if the entire database has been searched, the returned value of *lookup_handle* is the **lb_$default_lookup_handle** constant. This means a client can get all matching entries by repeating a lookup call, using the returned *lookup_handle* key, until the **lb_$default_lookup_handle** constant is returned, which indicates that the end of the database has been reached. For example, if an array processor client looks for array processors by doing type lookup, it can use the following Pascal code:

```
REPEAT
     lb_$lookup_type(array_tid, lookup_hndl, max_rslts, n_rslts,
         rslts, st);
UNTIL (lookup_hndl = lb_$default_lookup_handle);
```

Under normal conditions, repeated lookup calls obtain all matching entries in a database. However, some conditions, such as the databases being modified between lookup calls or the reception of two lookup calls by two different GLB databases, can cause entries to be skipped or duplicated. The client should be prepared to deal with missing or duplicated entries in the *results* array.

Converting between Host Names and Socket Addresses

The RPC runtime library also provides tools that convert between host names and socket addresses. For example, if you know the name of the destination host, the address family of the host, and the server port, you can make a call to the **rpc_$name_to_sockaddr** routine to obtain a socket address without using a Location Broker routine.

You *must* provide a value for the port number parameter of the **rpc_$name_to_sockaddr** routine. If you know that the server uses a well-known port, specify that port number. If you do not know the port number, specify **socket_$unspec_port**. When you use a socket address with a **socket_$unspec_port** to make a remote procedure call, the port number is determined at runtime. The RPC runtime library extracts a port number (if one was specified in the NIDL definition of the interface, from the *interface***$if_spec** variable). Otherwise, the call is sent to the forwarding port at the host.

The **rpc_$sockaddr_to_name** routine converts a socket address (such as the ones returned by Location Broker lookup routines) to a host name, which you can use in diagnostic output. For example, the **bankd.c** server program uses the following calls to identify its host and port number when it starts running:

```
rpc_$sockaddr_to_name(&saddr, slen, name, &namelen, &port, &st);
printf("(bankd) name=\"%.*s\", port=%d\n", namelen, name, port);
```

Using Handles That Are Not Fully Bound

You do not have to use a fully bound handle to make a remote procedure call. The handle can be bound to a host if the client knows only the host address but not the port the server uses. The handle can be unbound if the client does not know the location of the object.

Using Bound-to-Host Handles

When a program uses a bound-to-host handle to make a remote procedure call, the call is sent to the LLB forwarding port on that host. If the server for the requested interface has registered with the Location Broker, the LLB automatically forwards the call to the server's port. When the procedure call returns, the client's RPC runtime library then binds the handle to the server's port, and any subsequent calls are sent directly to the server.

Bound-to-host handles can be useful for clients of interfaces (such as remote shells) that are supported on many or all hosts. You do not need to do a Location Broker lookup operation to access the remote server. Only the host name or address is required. There is no need for the servers to register with the GLB, though each server must be registered in the LLB database on its host.

You can generate a bound-to-host handle by using a socket address with an unspecified port as an input parameter to an **rpc_$bind** or **rpc_$set_binding** routine, or by calling the **rpc_$clear_server_binding** routine on a fully bound handle.

Typically, an **rpc_$bind** or **rpc_$set_binding** routine is used to bind to a host if you used the **rpc_$name_to_sockaddr** routine to generate the socket address. For example, the following lines send a matrix multiplication call to the array processor server located at the host identified by `host_id`:

```
rpc_$name_to_sockaddr (host_id, hlen, socket_$unspec_port,
   socket_$internet, &saddr, slen, &st);
handle = rpc_$bind(&matrix_id, &saddr, slength, &st);
matrix$multiply(handle, a, b, result, $st);
```

The **rpc_$clear_server_binding** routine is particularly useful for error recovery. If a server fails and restarts using a new port, the client can reset the binding to the new port by calling the **rpc_$clear_server_binding** routine on the existing handle. The handle is then rebound when the server responds to the next client call.

Using Unbound Handles

When a program uses an unbound handle to make a remote procedure call, the call is broadcast on the local network. You can create an unbound handle by calling **rpc_$alloc_handle** to generate a new unbound handle, or by calling **rpc_$clear_binding** on an existing handle to clear the binding.

Generally, you should use an unbound handle only if you cannot use the Location Broker or the **rpc_$name_to_sockaddr** routine to determine the address of the host, because broadcast messages must be processed by all hosts that use the destination port.

If an operation has the **broadcast** attribute, it is always broadcast. The RPC runtime library automatically unbinds the handle after such an operation returns, so there is no need for the client to clear the binding before broadcasting again.

Determining the Binding

If a client application uses an unbound or bound-to-host handle to make a call, it could need to identify the server that responded (for example, for auditing). Because the handle is automatically bound to the responding server when the routine returns, you can call the **rpc_$inq_binding** routine to obtain the socket address represented by the returned handle, then call the **rpc_$sockaddr_to_name** routine to determine the server's address family, network address, and port. For example, the following code sequence generates an unbound handle, broadcasts for an available server, and then identifies the server:

```
my_handle = rpc_$alloc_handle(&my_object, socket_$internet,
   &status);
my_interface$find_server(my_handle, &status);
rpc_$inq_binding(my_handle, &server_sockaddr,
  &server_sockaddr_len, &status);
rpc_$sockaddr_to_name(&server_sockaddr, server_sockaddr_len,
   server_name, &server_name_len, &server_port, &status);
```

Generating and Managing RPC Handles

The RPC runtime library provides you with a flexible set of tools for creating and managing RPC handles. You can use RPC calls that allocate and bind handles simultaneously, or you can manage the creation and the binding state of handles separately.

The **rpc_$bind** routine simultaneously creates an RPC handle and binds it. A client should call the **rpc_$bind** routine if it accesses an interface on only one host. For example, the **bank.c** client program uses the following call to create the handle and bind it to the bank server:

```
h = rpc_$bind(&client_$if_spec, &client_id, &client_loc[0].saddr,
    bank_loc[0].saddr_len, &st);
```

The client then uses the *h* variable in the subsequent remote procedure call. Note that the client never frees the handle. It simply exits after getting a response from the server and printing the results.

The **rpc_$alloc_handle** routine creates an RPC handle that identifies an object, but not a specific server. The **rpc_$set_binding**, **rpc_$clear_binding**, and **rpc_$clear_server_binding** routines set and change the handle-binding state. These routines are useful in any application where the client can access an object on different hosts at different times. These routines allow the application to change the binding state without the expense of freeing and then creating the handle again.

For example, if an application must sequentially update all replicas of an object that are located on several different hosts, the client can do the following:

1. Make a call to the **rpc_$alloc_handle** routine to create a handle.
2. Determine the first server.
3. Make a call to the **rpc_$set_binding** routine to bind to the server.
4. Make the remote procedure call to update the replica.
5. Repeat steps 3 and 4, binding to servers on each host in sequence, to update all replicas.

Implementing Autobinding

If the client does not manage the RPC handles directly, but instead uses some other type of variable to identify the object, then you must write autobinding and auto-unbinding routines. The stub uses the binding routine to convert the client program's object identifier (the generic handle) into an RPC handle. Similarly, it uses the unbinding routine to free the RPC handle.

If an interface uses autobinding, the following occurs when the client makes a remote procedure call:

1. The client makes a remote procedure call (through the client switch) to the stub. The client provides a generic handle either as the first parameter of the routine (an explicit handle) or through a global variable (an implicit handle).
2. The stub calls the autobinding procedure, passing to it the generic handle.
3. The autobinding procedure returns an RPC handle to the stub.
4. The stub uses the RPC handle as a parameter to the **rpc_$sar** routine.
5. The **rpc_$sar** routine returns the server response to the stub.
6. The stub calls the auto-unbinding procedure, passing to it the RPC handle.
7. The auto-unbinding procedure returns a generic handle to the stub.
8. The stub returns to the client.

The generic handle can be a variable of any type that enables the autobinding procedure to identify the object and to generate a valid RPC handle. For example, if the generic handle is a UUID, the autobinding procedure can use the Location Broker to get the server socket address and call the **rpc_$bind** routine to create the RPC handle. In a remote file system interface, the generic handle can be a path name. If an interface supports only one object per host, the generic handle can be the host ID.

Routine Signatures

The only requirement or limitation imposed on the autobinding and auto-unbinding routines is the routine signature. The autobinding routine must be declared as follows:

- It must be named *type*_**bind**, where *type* is the type of the generic handle.
- It must take one input parameter, the generic handle, and have no output parameters.
- It must return the **handle_t** RPC handle.

The auto-unbinding routine must be declared as follows:

- It must be named *type*_**unbind**, where *type* is the type of the generic handle.
- It must take two input parameters, the generic handle and the **handle_t** RPC handle.

Routine Structure

The structure of the routines can vary considerably, depending on the type of the generic handle and the needs of the application. For example, if the generic handle is a path name, the autobinding procedure must somehow use the path name to determine the host socket address and object UUID before it can return the RPC handle.

Below is an example of autobinding. The routines in this program use UUIDs as generic handles and maintain a cache of handles to save the expense of using an **lb_$lookup_object** and an **rpc_$bind** routine each time the client makes a remote procedure call. This approach is particularly useful in applications where the client tends to make several calls to access the same object.

The autobinding routine, **uuid_$t_bind**, searches the cache for an RPC handle that matches the generic handle (the object UUID). If there is no matching handle in the cache, it calls the **lb_$lookup_object** routine to determine the location of the object and calls the **rpc_$bind** routine to construct a new handle. It uses the **rpc_$dup_handle** routine to return a copy of the handle.

Each handle in the cache has an associated reference count. When all copies have been freed, meaning that the binding is not in use, the original handle is kept available but is considered collectible. If its entry in the cache is needed for a new handle, it can be freed.

The auto-unbinding routine, **uuid_$t_unbind**, uses the **rpc_$free_handle** routine to free a copy of the RPC handle that matches the generic handle, then decrements the reference count of the handle.

Example of autobinding Routine

```
#include <idl/c/nbase.h>
#include <idl/c/lb.h>
#include <idl/c/uuid.h>
#include <sys/types.h>

#define MAX_ENTRIES 10

static struct db_entry {
/*Table mapping UUIDs into RPC handles */
    boolean  valid;           /* Is this entry valid? */
    uuid_$t  obj;             /* Object UUID */
    handle_t handle;          /* RPC handle for the object */
    unsigned short  refcnt;   /* # of references on this entry */
} uuid_db[MAX_ENTRIES];

handle_t uuid_$t_bind(object)
uuid_$t object;
{
    short i, invalid_i = -1, collectible_i = -1;
    lb_$entry_t lb_entry;
    int n_results;
    status_$t st;
    lb_$lookup_handle_t ehandle = lb_$default_lookup_handle;

/*
 * Scan the table for an entry that has a matching UUID.  If
 * we find one, return the handle that's stored there.  While
 * scanning, keep note of the last invalid (i.e. unused)
 * entry and the last collectible entry (i.e. one which has
 * an object/handle but isn't referenced by anyone).
 * If there is no match in the table, ask the LB for the
 * location.
 */

    for (i = 0; i < MAX_ENTRIES; i++) {
        struct db_entry *db = &uuid_db[i];
```

```
        if (! db->valid)
           invalid_i = i;
        else {
           if (bcmp(&db->obj, &object, sizeof object) == 0) {
               db->refcnt++;
               return (rpc_$dup_handle(db->handle, &st));
           }
           if (db->refcnt == 0)
               collectible_i = i;
        }
    }

  lb_$lookup_object(&object, &ehandle, 1, &n_results,
     &lb_entry, &st);
   if (st.all != status_$ok || n_results <= 0)
        abort();

 /*
  * Decide whether we have an entry to use.  Free the current
  * handle if we're collecting the entry.
  */

    if (invalid_i != -1)
        i = invalid_i;
    else if (collectible_i != -1) {
        i = collectible_i;
      rpc_$free_handle(uuid_db[i].handle, &st);
    }
    else
        abort();

    /*
     * Fill in the entry.  Make an RPC handle for the and return
     * it.
     */

   uuid_db[i].obj    = object;
   uuid_db[i].valid  = true;
   uuid_db[i].refcnt = 1;
   uuid_db[i].handle = rpc_$bind(&object, &lb_entry.saddr,
                             lb_entry.saddr_len, &st);
    if (st.all != status_$ok)
        abort();

  return (rpc_$dup_handle(uuid_db[i].handle,&st));
}

void uuid_$t_unbind(object, handle)
uuid_$t object;
handle_t handle;
{
    unsigned short i;
    status_$t st;

   /*
    * Scan the table looking for the handle.
    */

   for (i = 0; i < MAX_ENTRIES; i++) {
       struct db_entry *db = &uuid_db[i];
```

```
        if (db->valid && db->handle == handle) {
            rpc_$free_handle(handle, &st);
             db->refcnt--;
             return;
        }
    }
    abort();            /* Didn't find the handle in the table! */
}
```

Writing the Client Program

The source code you use to build a client program that makes remote procedure calls consists of the following elements:

- Header files, particularly the header file generated by the NIDL compiler
- The client application itself, which means the main client program and any other user-written routines that implement the application and call the remote procedures
- The client switch, generated from the interface definition by the NIDL compiler
- The client stub, generated from the interface definition by the NIDL compiler
- Any user-written routines that perform marshalling and autobinding.

If a client imports several interfaces, then the client source code must include the header file, client switch, client stub, and any marshalling and autobinding routines for each interface.

The following source files make up the client in a banking example:

- Header file: **bank.h**
- Client: **bankd.c** and **util.c**
- Client switch: **bank_cswtch.c**
- Client stub: **bank_cstub.c**.

The application code is contained in two files: **bank.c**, which contains the main program, and **util.c**, which contains utility routines that are used by both the client and the server. While a more complex program can be structured differently, NCS clients are usually built from at least the four elements listed above.

Handling Errors

In many ways, an RPC client handles errors in the same manner as a program that makes local procedure calls. However, there are also several types of errors that can be generated in the underlying RPC and communications mechanisms, and there are methods useful for detecting and handling such errors.

An RPC client can receive the following general classes of errors:

- Communications errors
- Server-failure errors
- Interface mismatch errors.

Communication Errors

Communications errors are errors that occur in the underlying communications mechanisms, resulting in the failure of a remote procedure call to reach the server or the failure of a server's response to reach the client. Communications errors are usually indicated by the **rpc_$comm_failure** status. To recover, a client can try the failed call again or try to find another server.

Server Failure Errors

Server failure errors are caused by a server crash. If the server process crashes while handling a remote procedure call, a fault is returned to the client indicating the failure. In this case, the error is signaled in the same manner as if the server had been locally linked with the client.

If the server fails and restarts between client RPC calls, the failure is usually indicated by an **rpc_$wrong_boot_time** status. If a server for another interface starts on the server host, using the same port number as the failed server, the client receives an **rpc_$wrong_if** status from the new server.

Recovery techniques depend on the type of call that is being made when the error gets signaled. If the server restarts and the application is connectionless (that is, the client and the server do not maintain any temporary state information between procedure calls), the client can call the **rpc_$clear_server_binding** routine to clear the handle port information. If the server did not restart, then the client must fully unbind, locate a working server, and rebind to the new server.

If the application does maintain some state between calls, the client must first clear the state (for example, by unwinding to the point at which it bound to the server), and then rebind with the restarted server or locate a new server.

Interface Mismatch Errors

Interface mismatch errors occur when the version of an interface used to generate the server stub is not identical to the version of the interface used to generate the client stub and switch. These errors are easily caught if you change the version number in the NIDL heading each time you change the interface definition. When version numbers do not match, the RPC runtime library signals a fault. If you do not change version numbers when you change the interface definition, the resulting errors are more difficult to detect.

Note: In most cases, a program cannot recover from mismatch errors. You must rebuild the out-of-date client or server.

Error-Handling Strategies for Remote Procedure Calls

The RPC runtime library always signals a fault if an error occurs when handling a remote procedure call. Therefore, you should set cleanup handlers around remote procedure calls to catch and handle any such faults.

A cleanup handler is set by calling the **pfm_$cleanup** library routine. If the operating system detects a fault while the cleanup handler is set, it does the following:

1. Unwinds the process stack to the most recent call to the **pfm_$cleanup** routine.
2. Releases the handler.
3. Returns from the **pfm_$cleanup** routine with the status value for the error that caused the fault.

Program execution then continues with the code that immediately follows the **pfm_$cleanup** routine. You usually follow the call with code that handles the fault. This code starts by testing the **pfm_$cleanup** return value so that the fault-handling statements are executed only when the handler is not set. Since in the normal case the handler is set, this code is executed only when a fault occurs.

A cleanup handler that is set at the start of the program returns the program to its start when a fault occurs. A cleanup handler can also be set before a particular call or procedure and then be released, using the **pfm_$rls_cleanup** routine, following the call. Use this technique to set a cleanup handler around remote procedure calls.

The example program below shows how to use a cleanup handler around a remote procedure call. The code fragment is based on the Global Location Broker source. This example routine allocates an RPC handle, calls the **pfm_$cleanup** routine (which returns a value of **pfm_$cleanup_set**), tests the return value, skips the error handling code if no fault occurs, and then makes a remote procedure call (**glb_$find_server**) that locates and binds to an available broker.

However, if a fault occurs in the **glb_$find_server** routine, the program unwinds back to the **pfm_$cleanup** routine, which returns the error status value. The error handling code then executes, testing whether the error was reported by the RPC runtime library or by some other module. If the error was signaled by the RPC runtime library, the code clears the handle and passes the status to the caller. Otherwise, it signals the fault again so that a previously set cleanup or fault handler can take the fault.

Note: The **pfm_$** routines are provided in the NCS RPC runtime library to enable you to implement fault handling routines. For a detailed discussion of the **pfm_$** routines, refer to pfm_$ Library Routines on page 5–86 and individual library routines.

```
#define RPC_ERROR(s) (((s).all & 0xffff0000) == rpc_$mod)
             /* macro to check whether a fault is an rpc error */

static int check_binding(status)
status_$t *status;
{
   pfm_$cleanup_rec crec;
   status_$t fst;
   int i;

   if (valid_binding)
      return TRUE;
   glb_$handle = rpc_$alloc_handle(&glb_$uid, socket_$internet,
      status);
   if (status->all == status_$ok)
      return FALSE;

   fst = pfm_$cleanup(crec);            /* set cleanup handler */
   if (fst.all != pfm_$cleanup_set) {    /* begin fault case */
      if (RPC_ERROR(fst)) {
         rpc_$free_handle(glb_$handle, status);
                                   /* if fault is an rpc error */
         *status = fst;             /* pass status up to caller */
      }
      else {                    /* if fault is not an rpc error */
         pfm_$signal(fst);  /* re-signal the fault to the next */
      }                            /* level cleanup handler */
   }
   else {                                /* begin normal case */
      glb_$find_server(glb_$handle, status);
      valid_binding = TRUE;
   }
   pfm_$rls_cleanup(crec, fst);     /* release cleanup handler */
   return valid_binding;
}
```

Writing the Server Program

The source code that you use to generate a server program consists of the following elements:

- Header files, particularly the header file generated by the NIDL compiler
- Server initialization code, which registers the server's interfaces
- Server stubs, one per interface, generated from the interface definition by the NIDL compiler
- User-written routines, if any, to perform marshalling
- Manager procedures, which implement the operations in the interfaces that the server exports. Manager procedures are independent of NCS and function exactly as in a local implementation.

The following source files make up the server in a banking example:

- Header file: **bank.h**
- Server initialization and manager: **bankd.c** and **util.c**
- Server stub: **bank_sstub.c**.

The initialization code and the manager are contained in two files: **bankd.c**, which contains the main program, and **util.c**, which contains utility routines that are used by both the client and the server. A more complex server can be structured differently.

The following figure shows the components of a server that exports two interfaces. The stub for one interface calls a user-written marshalling routine.

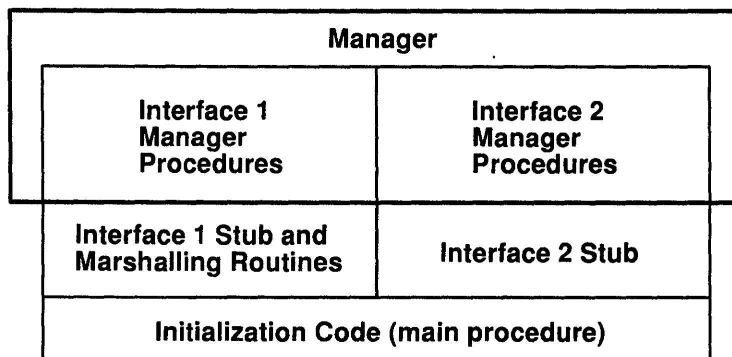


Figure 13. Components of a Server Exporting Two Interfaces

Writing the Server Main Procedure

The server initialization code is contained in the server main procedure, usually the Pascal PROGRAM or the C main procedure. This procedure does the following:

- Performs any type-specific or application-specific initialization.
- Establishes the sockets on which it will listen.
- Registers the server's interfaces with the RPC runtime library.
- Registers the server's objects and interfaces with the Location Broker.
- Establishes termination and fault-handling conditions.
- Begins listening for requests.

Most servers perform these chores in the order listed. A server must establish the sockets and register the interfaces with the RPC runtime library before it begins listening. The server begins listening last because the **rpc_$listen** routine normally does not return. A server should register with the Location Broker after it establishes the sockets.

Application-Specific Initialization

A program performs several application-specific tasks. In an example program called **bankd.c**, the program checks that the right number of input arguments are present. It checks the bank name input argument for validity, reads the bank database into memory from a file, and prints the bank name and the port number. The program does not print the bank name until it has established the socket and registered the interface. To get the socket name, the program calls the **rpc_$sockaddr_to_name** routine, using as input the address that was returned when the socket was established.

The server must call the **rpc_$use_family** or **rpc_$use_family_wk** routines once for each socket on which it listens. The **rpc_$use_family** routine assigns an available port for the socket, while the **rpc_$use_family_wk** routine assigns the well-known port that you specify. Do not use well-known ports unless necessary. A server can listen on more than one port simultaneously, although most servers do not need to listen on more than one port per address family.

The **bankd.c** main program uses the following routine to establish its socket:

```
rpc_$use_family(atoi(argv[1]), &saddr, &slen, &st);
```

Registering with the RPC Runtime Library

The server must call the **rpc_$register** routine once for each interface that it exports. Registration enables the RPC runtime library to call the required procedure when the server receives a packet requesting one of the operations in the interface.

The **bankd.c** main program uses the following routine to register its interface:

```
rpc_$register(&bank$if_spec, bank$server_epv, &st);
```

Remember that the NIDL compiler generates from an interface definition both an interface specifier, named *interface***$if_spec**, and a server entry point vector, named *interface***$server_epv**. The interface specifier is defined in the header file. The EPV is defined in the server stub.

Registering with the Location Broker

Most servers register their objects and interfaces with the Location Broker. Clients can then use **lb_$** lookup routines to locate objects. The server must use a separate **lb_$register** routine to register each possible combination of object, interface, and socket address. For example, a server should make six registration calls if all of the following is true:

- The server listens on an Internet socket.
- The server has one manager that exports two interfaces.
- The server manages three objects of the supported type.

Some object types, such as files or databases, may be numerous or may be frequently created and deleted. A server for such a type can register itself once for each type, interface, and address family combination, using **uuid_$nil** for the object UUID. Clients can use the Location Broker to look up only by type or by interface, but not by object UUID.

If a server runs on most or all hosts, it should register itself only with the Local Location Broker, and not with the Global Location Broker. In this case, clients can use a bound-to-host handle to access the interface. To register locally, specify **lb_$service_local** in the *flags* parameter of the **lb_$register** routine.

Because the socket address returned by the **rpc_$use_family** or **rpc_$use_family_wk** routine is an input argument to the **lb_$register** routine, interfaces must be registered with the Location Broker after sockets are established.

The **lb_$register** routine automatically replaces any existing registration that matches the specified object, type, interface, address family, and host. If there is no entry matching these values, the Location Broker adds a new entry. If a server terminates abnormally (without unregistering) and then restarts, the registration for the terminated server is replaced.

Note: It is the responsibility of the application developer to assign the type UUID and to use it consistently when registering and looking up objects in the Location Broker. Neither the RPC runtime library nor the NIDL compiler makes use of type identifiers.

The **bankd.c** main program uses the following routine to register its interface and object:

```
lb_$register(&bankID, &bank$uid, &bank$if_spec.id, 0, BankName,
     &saddr, slen, &BankEntry, &st);
```

Fault Handling and Termination

The initialization code also establishes any server fault handlers and signal catchers. For example, the **bankd.c** program includes the following lines to cause the **Terminate** routine to run whenever a hangup, interrupt, or quit signal occurs:

```
signal(SIGHUP, Terminate);
signal(SIGINT, Terminate);
signal(SIGQUIT, Terminate);
```

The routine is executed on both normal and abnormal termination.

The **Terminate** routine should unregister the server's objects and interfaces with the Location Broker so that clients do not try to access objects that are unavailable. Most other fault-handling and termination activities are application dependent.

The **Terminate** routine should be defined in the main program (**bankd.c**. in the example). The **Terminate** routine writes a message to the standard error output, unregisters the server with the Location Broker, writes the database to a disk file, and causes the program to exit.

In many cases, servers do not need cleanup handlers, which are necessary for clients. However, a fault or signal handler for asynchronous faults is recommended.

Writing the Manager Procedures

Although most of the remoteness of an NCS server is hidden in the server stub, the manager may require code to identify objects or to register objects with the Location Broker.

Identifying an Object

In an interface that uses explicit handles, the first parameter of any procedure must be the handle parameter. If the interface uses an implicit handle, the handle is not passed to the procedure. Handles for a manually bound interface have the type **handle_t**, while the handle type for an automatically bound interface is user defined.

A manager that supports multiple objects must be able to identify the particular object on which the client wishes to operate. Since the client has specified the object UUID in the handle, it is natural to pass the handle explicitly and have the manager obtain the object UUID from the handle. If the interface is manually bound, the manager can call the **rpc_$inq_object** routine to extract the object UUID from the **handle_t** handle. If the interface is automatically bound, the handle must be either the object UUID itself or some other data type from which the manager can determine the UUID.

For example, each manager routine in the **bankd.c** program calls the following **CheckObject** routine that gets the object UUID represented by the handle parameter and compares it with the UUID of the bank's database object:

```
static boolean CheckObject(h, st)
uuid_$th;
status_$t *st;
{
   if (bcmp(h, &BankID, sizeof(BankID))) {
        fprintf(stderr, "(bankd) Request for wrong bank!\n");
        st->all = -1;       /* "object not found" */
        return(false);
    }
    st->all = status_$ok;
    return(true);
}
```

Although, in this example, the server manages only one object, the code still shows how to determine object UUIDs from handles.

Registering Objects with the Location Broker

If a server manages only static objects, the server initialization code registers the objects with the Location Broker. However, if the server manages dynamic objects, such as files or databases, the manager routine that creates the objects can register newly created objects with the Location Broker by calling the **lb_$register** routine. Any routine that deletes an object must call the **lb_$unregister** routine to remove the registration.

Building an Application

Use the following steps to build an NCS application:

1. For each interface, run the NIDL compiler to generate header files and the source code for the server stub, the client stub, and the client switch.
2. For each interface, use the C compiler to generate object modules for the server stub, the client stub, and the client switch.
3. For each interface, compile the autobinding and marshalling routines, if any.
4. Compile the client application source to create the client object modules.
5. Bind the client switches, the client stubs, any autobinding routines, any marshalling routines, and the client application object modules to make the executable client.
6. Compile the server managers and initialization code to create the server object modules.
7. Bind the server stubs, any stub-marshalling routines, and the server object modules to make the executable server.

Remember that the client and the server must include the header or insert files for any library routines or types used, including any **rpc_$**, **pfm_$**, **lb_$**, or **uuid_$** routines or types. Similarly, any interface definition that includes predeclared system types should import the corresponding NIDL file. The NIDL files are located in the **/usr/include/idl** directory. The C header files are located in the **c** subdirectory.

If you are compiling in Pascal, create the Pascal insert files for each NIDL file in the **/usr/include/idl** directory using the following:

```
nidl -pascal -out pas -no_stubs -no_cpp
```

For example, to create the **base.ins.pas** file in the **/usr/include/idl/pas** subdirectory, type the following:

```
cd /usr/include/idl
mkdir pas
nidl -pascal -out pas -no_stubs -no_cpp
```

Note: A **makefile** function is located in the **/usr/lpp/ncs/examples/bank** directory that uses the **make** command to automate the building of the banking example.

Using C Syntax with NIDL

The C language syntax of the Network Interface Definition Language (NIDL) is a subset of ANSI C, with a few constructs added to express NCS remote procedure call semantics.

An NIDL interface definition must contain a heading and a body in the following form:

```
%c
[ interface_attribute_list ] interface identifier
{
body
}
```

The Heading

The interface definition heading consists of the following elements:

1. The syntax identifier, `%c`. This identifier specifies that the interface definition is written in the C syntax of NIDL. It must precede the remainder of the interface definition. The syntax identifier allows one compiler to parse both the C and the Pascal syntaxes of NIDL.
2. The *interface_attribute_list*. This list specifies characteristics of the interface. It must be enclosed in brackets, follow the syntax identifier, and precede the interface name.
3. The interface name. The name must be preceded by the interface attribute list and the **interface** keyword.

The interface attributes are described in detail later in this section.

The Body

The interface definition body follows the heading and is enclosed in braces. The body consists of one or more of these declarations:

- Import
- Constant
- Type
- Operation.

There must be at least one constant, type, or operation declaration. It is not sufficient to have an interface body containing only import declarations. Each declaration is terminated by a ; (semicolon).

The declarations are described in detail later in this section.

Comments

Comments in an NIDL definition are delimited as in the C language as follows:

```
/* all natural */
import "rpc.idl";  /* no preservatives */
```

Interface Attributes

The *interface_attribute_list* parameter is enclosed in brackets and includes one or more of the following elements, separated by commas:

- **uuid** (*uuid_string*)
- **port** (*port_identifier*, ...)
- **version** (*version_number*)
- **local**
- **implicit_handle** (*type identifier*).

The uuid Attribute

The **uuid** attribute designates the Universal Unique Identifier (UUID) assigned to the interface. No other object, interface, or type can be assigned this UUID. The **uuid** attribute is expressed by the keyword **uuid** followed in parentheses by the character-string representation of a UUID. (Refer to the description of the **uuid_$string_t** data type in uuid_$ Library Routines on page 5–94 for more information.)

The **/etc/ncs/uuid_gen** command generates character-string representations of UUIDs as shown in the following:

```
/etc/ncs/uuid_gen
```

This produces the following output:

```
334980380000.0d.00.00.37.27.00.00.00
```

The output of the **uuid_gen** command can be entered directly in the interface header. For example, the following interface definition heading includes a UUID attribute:

```
%c
[uuid (334980380000.0d.00.00.37.27.00.00.00)] interface
 my_interface
```

The port Attribute

The **port** attribute specifies the well-known port or ports on which servers that export the interface listen. Do not use this attribute in most cases. Instead, use the RPC runtime library to assign ports dynamically.

The **port** attribute has the following syntax:

```
port ( port_identifier1, ..., port_identifierN )
```

Each *port_identifier* field specifies an address family and a well-known port number. The *port_identifier* field takes the form *family*:[*port_number*]. The *family* parameter specifies the address family, and *port_number* indicates the well-known port.

In AIX, the only supported value for the *family* parameter is ip (Internet Protocols). However the following values for *family* are available in NIDL:

unspec	Unspecified
unix	Local to host (pipes, portals)
implink	ARPANET imp addresses
pup	Pup protocols (for example, BSP)
chaos	MIT CHAOS protocols
ns	XEROX NS protocols
nbs	NBS protocols
ecma	European computer manufacturers
datakit	Datakit protocols
ccitt	CCITT protocols (for example, X.25)
sna	IBM SNA
unspec2	Unspecified
dds	Domain DDS protocol
ip	Internet Protocols.

The following example assigns a well-known port for the Internet address family:

```
port(ip:[1230])
```

The version Attribute

The **version** attribute helps manage multiple versions of an interface. Its syntax is as follows:

```
version ( version_number )
```

The *version_number* parameter specifies an integer. You can increase the version number to indicate that the interface has changed in some upwardly compatible way. For example, if you were adding a new procedure to the `array_calc` interface, your heading could look like the following:

```
%c
[uuid(338b5f985000.0d.00.00.37.27.00.00.00), version (2)]
   interface array_calc
```

The local Attribute

The **local** attribute indicates that the NIDL definition does not declare any remote operations, which means only header files (no stub files) are generated. An interface definition heading must contain the **local** attribute or the **uuid** attribute. If you use **local**, the NIDL compiler ignores any other interface attributes.

The implicit_handle Attribute

The handle used to represent an object can be either an explicit parameter that is passed in each remote procedure call or an implicit global variable. The type of an implicit handle is determined by the **implicit_handle** attribute.

Explicit handles are the default handling technique. If you do not specify an **implicit_handle** attribute in the interface definition heading, the interface uses explicit handles, and each operation declaration must include a handle parameter as the first parameter of the declaration. Explicit handles enable the client to specify a different handle in each call it makes.

If you are using explicit handles, an operation declaration could look like the following:

```
my_proc (handle_t [in]   h,
              int [in]   high,
              int [in]   low,
              int [out]  average);
```

If you specify an **implicit_handle** attribute, the stub uses this handle to represent the remote object in all remote procedure calls, and no handle information is passed to the server. Do not include a handle parameter in the operation declarations.

The **implicit_handle** attribute has the following syntax:

```
implicit_handle ( type identifier )
```

The *type* and *identifier* parameters specify the type and name of the global variable to be used as an implicit handle.

If you use implicit handles for an array arithmetic interface, its definition heading could look like the following:

```
%c
[uuid(338b5f985000.0d.00.00.37.27.00.00.00),
implicit_handle(handle_t array_handle)]
interface array_calc
```

Import Declaration

The import declaration, which is similar to the C **#include** directive, specifies interface definition files that contain definitions for data types that the importing interface uses. It takes the following form:

```
import string ;
```

Each *string* parameter specifies the path name, enclosed in double quotation marks, of the file that you want to import. For example, the following declaration imports the base NIDL file:

```
import "/usr/include/idl/base.idl";
```

If an interface definition is imported, header files generated by the NIDL compiler contain C **#include** (or Pascal **%include**) directives for header files that correspond to the imported definitions. However, if an imported file contains operation declarations, stub procedures are *not* generated for these operations.

You can import interfaces written in either of the two NIDL syntaxes. The **import** declaration is idempotent. Importing an interface many times has the same effect as importing it once.

Constant Declaration

The constant declaration takes the following form:

```
const type_specifier identifier = integer | string|
   value_identifier ;
```

The *type_specifier* parameter specifies the data type of the constant being declared. The *identifier* parameter specifies the name of the constant. The *integer*, *string*, or *value_identifier* parameters indicate the value you are assigning to the constant. You can specify any previously defined constant as the *value_identifier* in a constant declaration. Constant expressions are not currently supported.

The C syntax of NIDL provides only **int** and **char** constants because it must be possible to compile any NIDL interface into languages (such as Pascal) that allow only simple constants.

Following are examples of constant declarations:

```
const int array_size = 100;
const char jsb = "Johann Sebastian Bach";
```

Types and Type Declarations

You can declare data types in type declarations and in the parameter lists of operation declarations. Note that some of the constructs used in type declarations can also be used in operation declarations.

A type declaration can appear in any of the following forms:

- A **struct** declaration
- A **union** declaration
- A **typedef** statement.

The type declaration takes the following form:

```
[ attributes ] type_specifier declaratory_list ;
```

Attributes

The following can be specified as attributes of types in a type declaration or of parameters in an operation declaration:

- **handle**
- **last_is** (*last*)
- **max_is** (*max*)
- **transmit_as** (*identifier*).

Enclose the list of attributes in brackets, with the attributes separated by commas.

The **handle** attribute can appear only in a type declaration, not in an operation declaration. You must specify this attribute when you declare a type which is to be used as a generic handle. You must also write autobinding and auto-unbinding routines to convert between the generic handle type and the RPC handle type. The following example declares that the **uuid_handle** type **uuid_$t** is to be used as a generic handle:

```
typedef [handle] uuid_handle uuid_$t;
```

The **last_is** and **max_is** attributes can appear in a **struct** declaration or in an operation declaration. The **transmit_as** attribute can appear in a **struct** declaration, in a **union** declaration, or in a **typedef** statement.

The **last_is** attribute allows the client to indicate dynamically the length of an array. This attribute informs the NIDL compiler that the variable named last holds, at runtime, the index of the last array element to be passed between the client and server. This value can be the index of the very last element, specifying that the entire array is to be passed, or it can be the index of an earlier element, specifying that only part of the array is to be passed.

You can apply the **last_is** attribute only to an array, and only use the attribute in a **struct** declaration or an operation declaration. In a structure declaration, the array is a member of the structure. In an operation declaration, the array is an operation parameter. In both cases, the **last_is** attribute specifies a last variable that holds the index of the array's last meaningful element. For example, the following **struct** declaration defines that the **plast** variable holds the index of the last element to be passed in the **pathname** array:

```
struct {
  int plast;
  [last_is (plast)] char pathname[];
  } open_pathname;
```

If you specify the **last_is** attribute for an array in an operation declaration, the last variable must have the same parameter attribute as the array. For instance, if the array is declared as an **in** parameter, the last must also be an **in** parameter.

In an operation declaration, you must use the **last_is** attribute for any open array, which is any array whose declaration does not include an explicit fixed length. For open arrays, the *last* variable controls the amount of data passed by the client and server stubs. For arrays of fixed length, **last_is** is optional. It optimizes performance when only part of the array needs to be passed.

If you specify the **last_is** attribute for an open array in a structure declaration, the last variable must be a member in the structure and must precede the array itself (as in the example).

The **max_is** attribute can be applied only to open arrays that are returned by the server. It enables the client to indicate the maximum size of the array to be returned. The specified *max* variable in a **max_is** attribute is the name of a variable that holds the maximum possible index of the array.

Like the **last_is** attribute, the **max_is** attribute can appear only in a **struct** declaration or in an operation declaration. In an operation declaration, the array and its *max* variable are parameters of the operation. In a structure declaration, the array and its *max* variable are members of the structure, and *max* must precede the array. For example, the following **struct** includes both **max_is** and **last_is** attributes for the **data_return** array:

```
typedef struct {
  int biggest_array;
  int array_end;
[max_is(biggest_array), last_is(array_end)] char data_return[];
} struct_with_open_array;
```

The server stub uses the *max* variable in a **max_is** attribute to determine how much storage space to allocate for an array. The *max* variable must have a value when the client makes the call. If you use the **max_is** attribute in an operation declaration, then *max* must have the **in** or **in out** parameter attribute. If you declare the array and its maximum length as members of a structure, the structure must be specified as an **in** or **in out** parameter in operation declarations.

The **max_is** attribute is not required for all open arrays. However, if you do not specify this parameter, then the array's last variable must be an **in** or **in out** parameter (or a member of an **in** or **in out** structure), and the server stub uses the last variable as if it had also been declared as a *max* variable.

The NIDL compiler cannot generate code to marshall and unmarshall pointers to pointers, pointers within structures, or pointers within unions. Therefore, if you use complex types such as trees and linked lists, you must provide routines that convert them into simpler transmissible types. The **transmit_as** attribute identifies a complex type and the transmissible type into which your routines convert it.

The *identifier* variable in a **transmit_as** attribute must be either an NIDL *type_specifier* or a previously defined type. It indicates the form in which the data is transmitted by the RPC runtime library. For example, the following **typedef** statements show the use of the **transmit_as** attribute for a binary tree:

```
typedef struct {
   data_t data;
   tree_t left;
   tree_t right;
} tree_node_t;
typedef [transmit_as(tree_xmit_t)] tree_node_t *tree_t;
```

Because **tree_t** is a pointer to **tree_node_t**, and **tree_node_t** contains pointers (to **tree_t**, no less), the NIDL Compiler cannot generate code to marshall variables of type **tree_t**. Instead, it generates code that calls user-written routines to convert between **tree_t** and **tree_xmit_t**, and the stubs actually transmit the data in **tree_xmit_t** format.

For any type that has a **transmit_as** attribute, you must provide the following routines and link them with the stubs:

*type*_**to__xmit_rep** — Converts from the complex type into the transmitted type.

*type*_**from_xmit_rep** — Converts from the transmitted type into the complex type.

*type*_**free** — Frees storage used by the server for the complex type.

*type*_**free_xmit_rep** — Frees storage used by the client for the transmitted type.

Declarator List

The declarator list specifies names of variables that have a particular type. The members of a declarator list can be simple names, pointer names, or array names. If your list includes multiple names, separate the names with commas as shown in the following:

```
char     input_array[10];        /* Single identifier */
float    open, high, low, close; /* Multiple identifiers */
```

To specify a pointer, precede the name with an * (asterisk) as shown in the following:

```
long int *point_to_int;
```

To specify an array, put brackets after the name. Although it is not required, you can put the array size or an asterisk inside the brackets. If you put an asterisk or if you put nothing, you are declaring an open array that has a length that is not known until runtime, and you must use the **last_is** type attribute in its type declaration. Array subscripts start at 0. The following example of a structure (**struct**) declaration includes two arrays:

```
struct { char last_name[15];
         int  matrix[*]; }
```

If a **struct** declaration contains an open array, the array must be the last member. A **struct** declaration containing an open array cannot be returned by an operation as its value or as a parameter. A union cannot contain an open array.

Multidimensional arrays are declared with consecutive pairs of brackets, as in the C language. An example follows:

```
int two_by_four [2][4];
```

Only the first dimension of a multidimensional array can be unspecified:

```
int n_by_four [][4]; /* this is valid */
int two_by_n [2][]; /* this is NOT valid */
```

The typedef Constructor

As in the C language, the NIDL **typedef** statement allows you to create a synonym for a data type. Its syntax takes the following form:

```
typedef type_declaration;
```

For instance, the following **typedef** statement defines an integer of the **long_int** type to be a synonym for regular integer, **int**:

```
typedef int long_int;
```

Typically, a **typedef** statement is used to create a distinct name for a **struct** declaration, as shown in the following:

```
typedef struct { char month[3];
                 int  day, year;
               } birthday;
```

Type Specifiers

The type specifier portion of a type declaration can specify any of the following types:

- Simple types
- Constructed types
- **handle_t**
- Types defined with **typedef** statements (as defined in the previous section).

Simple Types

NIDL supports the following simple data types:

- Integers, which include the following:
  - **int**
  - **long int**
  - **short int**
  - **unsigned int**
  - **unsigned long int**
  - **unsigned short int**

  The **int** keyword is optional after any of the five other integer type names. For example, **long** and **long int** are synonymous. The **int**, **long**, **unsigned**, and **unsigned long** types are represented in 32 bits. A **short** or **unsigned short** type is represented in 16 bits.

- Floating-point types, which include the following:
  - **float**, which is represented in 32 bits
  - **double**, which is represented in 64 bits.

- Character types, which include the following:
  - **char**
  - **unsigned char**

  NIDL does not support a signed character. If you specify **char**, the NIDL compiler treats the type as **unsigned char**.

- The **boolean** type. Following convention, a value of 0 (zero) means false, and a nonzero value means true.

- The **byte** type. This is an 8-bit integer that is not converted when transmitted by the RPC runtime mechanism. To protect data of any type from conversion, you can write routines to convert that type into an array of byte.

- The **void** type. This type is most often used for a function that does not return a value.

- Enumerated types, which include the following:

  ```
  enum { identifier1, . . . , identifierN }
  short enum { identifier1, . . . , identifierN }
  ```

  The enumerated types provide names for integers. An `enum` type is a 32-bit integer, and a `short enum` type is a 16-bit integer. The compiler assigns integer values, beginning at `0`, to `enum` identifiers based on their order in the list. For example, in the following `enum` declaration:

  ```
  enum {John, Paul, George, Ringo} beatles_members;
  ```

  `John` gets the value 0, `Paul` gets 1, `George` gets 2, and `Ringo` gets 3.

Constructed Types

NIDL also supports the following constructed data types:

- Sets, which provide names for bits within single integers, starting with the least significant bit of a 16-bit integer. This is similar to **enum** types, which provide names for integers. A set takes the following form:

  ```
  set { identifier1, . . . , identifierN }
  ```

  For example, in the following **set** declaration, `Steinhardt` represents the value of bit 0 in an integer, `Dalley` represents bit 1, `Tree` represents bit 2, and `Soyer` represents bit 3:

  ```
  set {Steinhardt, Dalley, Tree, Soyer} guarneri_quartet_members;
  ```

- Strings, which take the following form:

  ```
  string0 [ length ]
  ```

  A string0 is a C-style null-terminated string, which is a character array whose last element is the C null character, \0. The *length* parameter indicates the maximum length of the string.

- Structures that are named in **typedef** statements as shown in the following:

  ```
  struct { type_declaration1; . . . ; type_declarationN; }
  ```

  This type differs slightly from its C counterpart. An NIDL **struct** statement cannot contain pointers unless it is to be marshalled by user-written routines. NIDL does not allow structures to have tags.

- Unions, which take the following form:

  ```
  union switch ( simple_type_specifier identifier )
  { component1;
  .
  .
  .
  componentN; }
  ```

  The *simple_type_specifier* parameter specifies one of the simple data types described in the previous section. The *identifier* parameter specifies the name of a discriminating variable, and each *component* variable consists of the following:

  ```
  case constant : type_declaration
  ```

  This type differs considerably from its C counterpart. In NIDL, unions must be discriminated. That is, in the union header you must specify a discriminating variable and its type. The generated stubs use this variable to determine which member is relevant for a routine. The NIDL **union** looks like a cross between the C **union** and **switch** statements.

  In an NIDL **union**, the *declarator_list* portion of the *type_declaration* parameter is limited to one name. A **union**, like a **struct** (structure), cannot have a tag, nor can it contain pointers unless it is to be marshalled by user-written routines.

The *type_declaration* parameter in a case clause is optional. Its omission indicates that several cases all take the same declarator. For example, the following defines an enumerated data type, **stats**, and uses the names in **stats** to discriminate a union:

```
enum {atbats, hits, avg} stats;
union switch ( enum stats )
   { case atbats   :
   case hits      : int total;
   case avg       : float average; } stats_union;
```

The handle_t Type

The **handle_t** type indicates that the variable is a handle type meaningful to the RPC runtime mechanism. If you specify this type for the first parameter of each operation in an explicitly bound interface or for the handle variable of an implicitly bound interface, the interface uses manual binding. The client application must manage the binding state of the handle.

The Operation Declaration

The operation declaration in NIDL is similar to a function heading in the C language. The syntax takes the following form:

```
[ operation_attributes ] type_specifier declarator (param_list);
```

The optional *operation_attributes* parameters specify calling semantics. The following attributes apply:

- **idempotent**
- **broadcast**
- **maybe**

If you specify more than one of these, separate the attribute names with commas. Enclose the entire list in square brackets.

The idempotent Attribute

By default, the RPC runtime software provides at most once calling semantics. It ensures that a remote procedure, when called once, is not executed more than once. The **idempotent** attribute overrides this default for a particular operation, allowing the operation to be executed as many times as is necessary to get a response. This reduces overhead in the RPC mechanism, improving performance. Use the **idempotent** attribute only if the operation can safely be executed multiple times. For instance, an operation that simply reads a value is idempotent, while one that increments a value is not.

The broadcast Attribute

The **broadcast** attribute specifies that the client's RPC runtime software always broadcasts this operation to all hosts on the local network. When an operation with the **broadcast** attribute is called, the handle is automatically unbound before the remote procedure call is issued.

The maybe Attribute

The **maybe** attribute specifies that the caller does not expect any response and that the RPC runtime system need not guarantee delivery of the call. Use this attribute to post a notification whose receipt is not crucial. Operations with the **maybe** attribute cannot have any output parameters.

Simple Type Specifiers and Declarators

In any operation declaration, you must give the declaration a data type and a name.The *type_specifier* parameter in the operation declaration is the data type returned by the operation. It can be any scalar or previously named data type, but it cannot be a pointer. For example, if the operation returns a long integer, specify **long** as the *type_specifier* parameter. Specify **void** if the operation does not return. If you omit the *type_specifier* parameter, the operation must return an **int** value.

Note: The declarator is the name of the operation.

The Parameter List

The parameters of an operation appear in the *param_list* field. The list takes the following form:

```
[ attributes1 ] type_specifier1 [ param_attributes1 ]
    declarator1 , . . . ,
[ attributesN ] type_specifierN [ param_attributesN ] declaratorN
```

If you have more than one entry in a *param_list* field, separate the entries with commas. In an operation declaration, you can use the **last_is** and **max_is** attributes. The *type_specifier* field specifies the data type of the parameter. It can be any of the simple data types.

Note: If an interface uses explicit handles (the interface definition heading does not specify an **implicit_handle** attribute), the first parameter in *param_list* must be the explicit handle. If the interface also uses manual binding, this parameter must have the **handle_t** type.

The *param_attributes* in the list specify the direction in which the parameter is passed using one of the following:

in	Passed from client to server.
out	Passed from server to client.
in, out	Passed both ways.

To specify that the parameter is passed by reference from the client to the server,, precede the *declarator* parameter with an & (ampersand), the address operator. This construct is equivalent to the **in ref** attributes in the Pascal syntax of NIDL. It is typically used when the application software is implemented in Pascal.

Note: The *declarator* parameter specifies the name of each parameter.

Operation Declaration Example

The following example shows the elements in an operation declaration:

```
[broadcast] int array$add ( handle_t [in]  bind_var,
                            int      [in]  array1[10][10],
                            int      [in]  array2[10][10],
                            int      [out] result );
```

Using Pascal Syntax with NIDL

The Pascal syntax of the Network Interface Definition Language (NIDL) is a subset of Pascal, with a few constructs added to express NCS remote procedure call semantics.

An NIDL interface definition must contain a heading and a body in the following form:

```
%pascal
[ interface_attribute_list ] interface identifier ;
body
end;
```

The Heading

The interface definition heading consists of the following elements:

- 1The syntax identifier, **%pascal**. This identifier specifies that the interface definition is written in the Pascal syntax of NIDL, and must precede the remainder of the interface definition. The syntax identifier allows one compiler to parse both the C and the Pascal syntaxes of NIDL.
- 2The *interface_attribute_list*. This list specifies characteristics of the interface. The list must be enclosed in brackets, follow the syntax identifier, and precede the interface name.
- 3The interface name. The name specified by the *identifier* parameter must be preceded by the interface attribute list and the **interface** keyword, and be followed by a semicolon.

The Body

The interface definition body follows the heading and is enclosed in braces. The body consists of one or more of these declarations:

- Import declaration
- Constant declaration
- Type declaration
- Operation declaration.

There must be at least one constant, type, or operation declaration. It is not sufficient to have an interface body containing only import declarations. Each declaration is terminated by a ; (semicolon).

Comments

Comments in an NIDL definition are delimited as in the Pascal language, as follows:

```
{ all natural }
import 'rpc.idl';  { no preservatives }
```

Interface Attributes

The *interface_attribute_list* field is enclosed in brackets and includes one or more of the following elements, separated by commas:

- **uuid** (*uuid_string*)
- **port** (*port_identifier*, ...)
- **version** (*version_number*)
- **local**
- **implicit_handle** (*identifier* : *type*).

The uuid Attribute

The **uuid** attribute designates the Universal Unique Identifier (UUID) assigned to the interface. No other object, interface, or type may be assigned this UUID. The **uuid** attribute is expressed by the keyword **uuid** followed in parentheses by the character-string representation of a UUID. (Refer to the description of the **uuid_$string_t** data type in uuuid_$ Library Routines on page 5–94 for more information.)

The **/etc/ncs/uuid_gen** command generates character-string representations of UUIDs as shown in the following:

```
/etc/ncs/uuid_gen
334980380000.0d.00.00.37.27.00.00.00
```

The output of the **uuid_gen** command can be entered directly in the interface header. For example, the following interface definition heading includes a UUID attribute:

```
%pascal
[uuid (334980380000.0d.00.00.37.27.00.00.00)] interface
 my_interface;
```

The port Attribute

The **port** attribute specifies the well-known port or ports on which servers that export the interface listen. In most cases, do not use this attribute. Instead, use the RPC runtime library to assign ports dynamically.

The **port** attribute has the following syntax:

```
port ( port_identifier1, ..., port_identifierN )
```

Each *port_identifier* field specifies an address family and a well-known port number. The *port_identifier* field takes the form *family*:[*port_number*]. The *family* parameter specifies the address family and the *port_number* parameter indicates the well-known port.

In AIX, the only supported value for the family parameter is ip (Internet Protocols). However the following values for *family* are available in NIDL:

unspec	Unspecified
unix	Local to host (pipes, portals)
ip	Internet Protocols
implink	ARPANET imp addresses
pup	Pup protocols (for example, BSP)
chaos	MIT CHAOS protocols
ns	XEROX NS protocols
nbs	NBS protocols
ecma	European computer manufacturers
datakit	Datakit protocols
ccitt	CCITT protocols (for example, X.25)
sna	IBM SNA
unspec2	Unspecified
dds	Domain DDS protocol.

The following example assigns a well-known port for the Internet address family:

```
port(ip:[1230])
```

The version Attribute

The **version** attribute helps manage multiple versions of an interface. Its syntax is as follows:

```
version ( version_number )
```

The *version_number* parameter specifies an integer. You can increase the version number to indicate that the interface has changed in some upwardly compatible way. For example, if you were adding a new procedure to the **array_calc** interface, your heading could look like the following:

```
%pascal
[uuid(338b5f985000.0d.00.00.37.27.00.00.00), version (2)]
interface array_calc
```

The local Attribute

The **local** attribute indicates that the NIDL definition does not declare any remote operations. Only header files (no stub files) are generated. An interface definition heading must contain the **local** attribute or the **uuid** attribute. If you use **local**, the NIDL compiler ignores any other interface attributes.

The implicit_handle Attribute

The handle that is used to represent an object can be either an explicit parameter that is passed in each remote procedure call or an implicit global variable. The type of an implicit handle is determined by the **implicit_handle** attribute.

Explicit handles are the default handling technique. If you do not specify an **implicit_handle** attribute in the interface definition heading, the interface uses explicit handles, and each operation declaration must include a handle parameter as the first parameter of the declaration. Explicit handles enable the client to specify a different handle in each call it makes.

If you are using explicit handles, an operation declaration could look like the following:

```
procedure my_proc(
           in  h: handle_t;
           in  high,low: int;
           out average: int
           );
```

If you specify an **implicit_handle** attribute, the stub uses this handle to represent the remote object in all remote procedure calls, and no handle information is passed to the server. You should not include a handle parameter in the operation declarations.

The **implicit_handle** attribute has the following syntax:

```
implicit_handle ( identifier: type )
```

The *type* and *identifier* parameters specify the type and name of the global variable to be used as an implicit handle.

If you use implicit handles for an array arithmetic interface, its definition heading could look like the following:

```
%pascal
[uuid(338b5f985000.0d.00.00.37.27.00.00.00),
                implicit_handle(array_handle: handle_t)]
interface array_calc;
```

Import Declaration

The import declaration, which is similar to the Pascal **%include** directive, specifies interface definition files that contain definitions for data types that the importing interface uses. It takes the following form:

```
import string1, ..., stringN ;
```

Each *string* parameter specifies the path name, enclosed in single quotation marks, of a file that you want to import. If you list several files, separate them with commas. For example, the following declaration imports two NIDL files:

```
import '/usr/include/idl/base.idl',
       '/usr/include/idl/rpc.idl';
```

If an interface definition is imported, header files generated by the NIDL compiler contain C **#include** (or Pascal **%include**) directives for header files that correspond to the imported definitions. However, if an imported file contains operation declarations, stub procedures are not generated for these operations.

You may import interfaces written in either of the two NIDL syntaxes. The **import** declaration is idempotent. Importing an interface many times has the same effect as importing it once.

Constant Declaration

The constant declaration takes the following form:

```
const
   identifier1 = integer1 | string1 | value_identifier1 ;
     . . . ;
   identifierN = integerN | stringN | value_identifierN ;
```

The *identifier* parameter specifies the name of the constant. The *integer*, *string*, or *value_identifier* parameters specify the value you are assigning to the constant. You can specify any previously defined constant as the *value_identifier* parameter in a constant declaration. Several constants may be declared under one **const** keyword. Each declaration should be terminated by a ; (semicolon). Constant expressions are not currently supported.

The following example declares two constants:

```
const
    array_size = 100;
    jsb        = 'johann sebastian bach';
```

Types and Type Declarations

You can declare data types in type declarations and in the parameter lists of operation declarations. Note that some of the constructs used in type declarations can also be used in operation declarations.

A type declaration takes the following form:

```
type
   identifier1 = type_descript1 | ^ptr_type1 ;
    . . . ;
   identifierN = type_descriptN | ^ptr_typeN ;
```

The *identifier* parameter specifies the name you are assigning to the data type.

You must provide either a *type_descript* or a *ptr_type* parameter. A *type_descript* parameter describes attributes of the type, and has the following syntax:

```
[ attributes ] type_specifier
```

A *ptr_type* parameter, preceded by a ^ (circumflex), names a pointer type.

Note: There is an important difference between ordinary Pascal and the Pascal syntax of NIDL. Standard Pascal uses the **type** keyword to define data types and the **var** keyword to declare variables having those types. NIDL does not have a **var** keyword. Variables are declared in the parameter lists of operations.

Attributes

The following attributes can appear in a type declaration or as part of an operation declaration:

- **handle**
- **last_is** (*last)*
- **max_is** (*max*)
- **transmit_as** (*identifier*).

The attributes should be enclosed in brackets, with the attributes separated by commas.

The **handle** attribute can appear only in a type declaration, not in an operation declaration. You must specify this attribute when you declare a type which is to be used as a generic handle. You must also write autobinding and auto-unbinding routines to convert between the generic handle type and the RPC handle type. The following example declares that the **uuid_handle** type (a **uuid_$t** data type) is to be used as a generic handle:

```
type uuid_handle = [handle] uuid_$t;
```

The **last_is** attribute allows the client to indicate dynamically the length of an array. This attribute informs the NIDL compiler that the variable named last holds, at runtime, the index of the last array element to be passed between the client and server. This value can be the index of the very last element, specifying that the entire array is to be passed, or it can be the index of an earlier element, specifying that only part of the array is to be passed.

You can apply the **last_is** attribute only to an array. You can use the attribute only in a **record** declaration or an operation declaration. In a record declaration, the array is a field in the record. In an operation declaration, the array is an operation parameter. In both cases, the **last_is** attribute specifies a last variable that holds the index of the array's last meaningful element. For example, the following **record** declaration defines that the **plast** variable holds the index of the last element to be passed in the **pathname** array:

```
type
     open_pathname = record
              plast:      integer;
             pathname:  [last_is (plast)] char;
      end;
```

If you specify the **last_is** attribute for an array in an operation declaration, the last variable must have the same parameter attribute as the array. For instance, if the array is declared as an **in** parameter, the last variable must also be an **in** parameter.

In an operation declaration, you must use the **last_is** attribute for any **open array**, which is any array whose declaration does not include an explicit fixed length. For open arrays, the last variable controls the amount of data passed by the client and server stubs. For arrays of fixed length, the **last_is** parameter is optional. It optimizes performance when only part of the array needs to be passed.

If you specify the **last_is** attribute for an open array in a record declaration, the last variable must be a field in the record and must precede the array itself (as in the preceding example).

The **max_is** attribute can be applied only to open arrays that are returned by the server. It enables the client to indicate the maximum size of the array to be returned. The specified *max* variable in a **max_is** attribute is the name of a variable that holds the maximum possible index of the array.

Like the **last_is** attribute, the **max_is** attribute can appear only in a record declaration or in an operation declaration. In an operation declaration, the array and its *max* variable are parameters of the operation. In a record declaration, the array and its *max* variable are fields in the record, and the *max* variable must precede the array. For example, the following record includes both **max_is** and **last_is** attributes for the **data_return** array:

```
type rec_with_open_array = record
           biggest_array:         integer;
           array_end:  integer;
          [max_is(biggest_array), last_is(array_end)]
                   data_return:
           array[1.. *] of char;
      end;
```

The server stub uses the *max* variable in a **max_is** attribute to determine how much storage space to allocate for an array. The *max* variable must have a value when the client makes the call. If you use the **max_is** attribute in an operation declaration, then *max* must have the **in** or **in out** parameter attribute. If you declare the array and its maximum length as fields in a record, the record must be specified as an **in** or **in out** parameter in operation declarations.

The **max_is** attribute is not required for all open arrays. However, if you do not specify this parameter, then the array's last variable must be an **in** or **in out** parameter (or a field in an **in** or **in out** record). The server stub then uses the last variable as if it had also been declared as a *max* variable.

The NIDL compiler cannot generate code to marshall and unmarshall pointers to pointers, pointers within structures, or pointers within unions. Therefore, if you use complex types such as trees and linked lists, you must provide routines that convert them into simpler transmissible types. The **transmit_as** attribute identifies a complex type and the transmissible type into which your routines can convert it.

The *identifier* variable in a **transmit_as** attribute must be either a NIDL *type_specifier* parameter or a previously defined type that indicates the form in which the data is transmitted by the RPC runtime library. The following example shows the use of the **transmit_as** attribute for a binary tree:

```
type
  tree_t = [transmit_as(tree_xmit_t)] ^tree_node_t;
   tree_node_t = record
          data  : data_t;
          left,
          right : tree_t;
           end;
```

Because the **tree_t** variable is a pointer to **tree_node_t**, and **tree_node_t** contains pointers (to **tree_t**, no less), the NIDL compiler cannot generate code to marshall variables of type **tree_t**. Instead, it generates code that calls user-written routines to convert between **tree_t** and **tree_xmit_t**, and the stubs actually transmit the data in **tree_xmit_t** format. For any type that has a **transmit_as** attribute, you must provide the following routines and link them with the stubs:

*type*_**to__xmit_rep**	Converts from the complex type into the transmitted type.
*type*_**from_xmit_rep**	Converts from the transmitted type into the complex type.
*type*_**free**	Frees storage used by the server for the complex type.
*type*_**free_xmit_rep**	Frees storage used by the client for the transmitted type.

Type Specifiers

The *type_specifier* portion of a type declaration can specify any of the following types:

- Simple types
- Structured types
- **handle_t** type
- Types defined using **type** statements.

Simple Types

NIDL supports the following simple data types:

- Integers, which include the following:
  - **integer**
  - **integer32**
  - **unsigned**
  - **unsigned32**

  The **integer** and **unsigned** types are represented in 16 bits. The **integer32** and **unsigned32** types are represented in 32 bits.

- Floating-point types, which include the following:
  - **real**, which is represented in 32 bits
  - **double**, which is represented in 64 bits.
- The **char** type.
- The **boolean** type.
- The **byte** type. This is an 8-bit integer that is not converted when transmitted by the RPC runtime mechanism. To protect data of any type from conversion, you can write routines to convert that type into an array of byte.
- Enumerations, which provide names for 16-bit integers. A list of identifiers is provided in parentheses, with the identifiers separated by commas. The compiler assigns integer values, beginning at 0, to `enum` identifiers based on their order in the list. For example, in the following `enum` declaration:

  ```
  enum {John, Paul, George, Ringo} beatles_members;
  ```

  `John` gets the value 0, `Paul` gets 1, `George` gets 2, and `Ringo` gets 3.
- Subranges, which specify a subrange of integers or of any previously defined enumeration. Provide the lower and upper limits of the subrange separated by two periods. The following example declares a subrange:

  ```
  small_ints = 1..100;
  ```

Structured Data Types

NIDL also supports the following structured data types:

- Arrays, which take the following form:

  ```
  array [ simple_type1, ..., simple_typeN ] of type_descript
  ```

  The *simple_type* parameter indicates the index values allowed for each dimension of the array. This parameter can be any simple data type but is most often a subrange. The *type_descript* parameter can be any simple data type or previously defined structured type. For example, this array includes two dimensions, both of which are specified with the following subranges:

  ```
  matrix = array[1..10, 1..5] of integer;
  ```

  To specify that the first dimension of an array is of varying length, use an * (asterisk) as the upper limit of the index value. Note that only the first dimension of an array can be varying. The following example shows how to use the * (asterisk):

  ```
  daily_temps = array[1..*, 1..12] of real; {this is valid }
  two_by_n = array[1..2, 1..*] of real; {this is NOT valid }
  ```

  If you declare an open array (an array of varying length) as a field in a record or as a parameter in an operation, you must use the **last_is** attribute, as described earlier.
- Strings, which take the following form:

  ```
  string0 [  length  ]
  ```

  A string0 is a C-style null-terminated string, which is a character array whose last element is the C null character, \0. The *length* variable indicates the maximum length of the string.

- Sets, which provide names for bits within single integers, starting with the least significant bit of a 16-bit integer. Sets take the following form:

  ```
  set of enumerated_type | ( identifier1,..., identifierN )
  ```

  A set is similar to an enumeration, which provides names for integers. The following example declares an enumeration called **beatles_members** and a set called **beatles_set**:

  ```
  beatles_members = (John, Paul, George, Ringo);
  beatles_set = set of beatles_members;
  ```

  In this set, `John` represents the value of bit 0 in an integer, `Paul` represents bit 1, `George` represents bit 2, and `Ringo` represents bit 3.

  Alternatively, you can define an enumerated type and declare a set in one step. For example:

  ```
  guarneri_quartet_set = set of (Steinhardt, Dalley, Tree,
     Soyer);
  ```

- Records, which take the following form:

  ```
  record
     member_list1  :  type_descript1  ;
     . . .
    member_listN  :  type_descriptN  ;
  end
  ```

  The *type_descript* parameter can be any simple data type or previously defined structured data type. The *member_list* parameter consists of one or more identifiers, separated by commas. The following example defines a record:

  ```
  type
     date_type = array[1..8] of char;
     weather = record
       date : date_type;
       hi_temp, low_temp, rain : real;
       pressure : double;
      end;
  ```

- Variant records, which take the following form:

  ```
  record case tag : simple_type of
     union_component1 ;
      . . . ;
     union_componentN ;
  end
  ```

  The *tag* parameter is analogous to the tag field in the case portion of a Pascal variant record. The *simple_type* parameter specifies a simple data type specifying the type of *tag*. Each *union_component* parameter has the following form:

  ```
  const_exp1, ... , const_expN : ( field1 ; ... ; fieldN )
  ```

  In a *union_component*, the *const_exp* variable can be an integer, an enumeration value, or a previously defined constant. Each *field* has the following form:

  ```
  identifier1, . . . , identifierN : type_descript
  ```

The *identifier* parameter specifies the name of a field in the record, and the *type_descript* component specifies the data type of the field. The *type_descript* can be any simple data type or previously defined structured type. The following example shows how a variant record is defined:

```
my_union = record case result : integer of
    1   : (       a,b,c : real);
    2,3 : (           d : boolean);
    4   : (         all : array[1..4] of char);
    5   : ( first_half : array[1..2] of char;
           second_half : array[1..2] of char);
end;
```

The handle_t Type

The **handle_t** type indicates that the variable is a handle type meaningful to the RPC runtime mechanism. If you specify this type for the first parameter of each operation in an explicitly bound interface or for the handle variable of an implicitly bound interface, the interface uses manual binding. The client application must manage the binding state of the handle.

The Operation Declaration

The operation declaration in NIDL is analogous to a procedure or function heading in Pascal. It can take two forms:

```
[ operation_attributes ] procedure identifier ( param_list  )

[ operation_attributes ] function identifier ( param_list  )
    :type_descript
```

The optional *operation_attributes* parameters specify calling semantics. The following attributes apply:

- **idempotent**
- **broadcast**
- **maybe**

If you specify more than one of these, separate the attribute names with commas. Enclose the entire list in square brackets.

The idempotent Attribute

By default, the RPC runtime software provides at most once calling semantics. This ensures that a remote procedure, when called once, is not executed more than once. The **idempotent** attribute overrides this default for a particular operation, allowing the operation to be executed as many times as necessary to get a response. In other words, the **idempotent** attribute specifies at least once semantics. This reduces overhead in the RPC mechanism, improving performance. Use **idempotent** only if the operation can safely be executed multiple times. For example, an operation that simply reads a value is idempotent, while one that increments a value is not.

The broadcast Attribute

The **broadcast** attribute specifies that the client's RPC runtime software always broadcasts this operation to all hosts on the local network. When an operation with the **broadcast** attribute is called, the handle is automatically unbound before the remote procedure call is issued.

The maybe Attribute

The **maybe** attribute specifies that the caller does not expect any response and that the RPC runtime system need not guarantee delivery of the call. Use this attribute to post a notification whose receipt is not crucial. Operations with the **maybe** attribute cannot have any output parameters.

Naming and Typing the Operation

In any operation declaration, you must specify whether the operation is a procedure or a function, and you must name it with an *identifier* variable.

If you specify that the operation is a function, you must also specify a *type_descript*, declaring the data type that the function returns. The *type_descript* can be any simple data type or previously named data type, but it cannot be a pointer.

The Parameter List

The parameters of an operation appear in the *param_list* field. The list takes the following form:

```
param_class1 name_list1  :  attributes1 type_name1  ;
    . . . ;
param_classN name_listN  :  attributesN type_nameN
```

If you have more than one entry in a *param_list* field, separate the entries with semicolons. Each *name_list* field lists parameters of the operation, separated by commas. The *type_name* parameter specifies any simple data type or previously named data type.

Note: If an interface uses explicit handles (if the interface definition heading does not specify an **implicit_handle** attribute), the first parameter in the *param_list* field must be the explicit handle. If the interface also uses manual binding, this parameter must have the **handle_t** type.

In an operation declaration, you can use the **last_is** and **max_is** attributes.

The *param_class* field specifies the direction and manner in which the parameter is passed, from the following choices:

in	Passed from client to server
out	Passed from server to client
in out	Passed both ways
in ref	Passed by reference from client to server.

Note: Users of Pascal should note that values larger than 32 bits must be passed by reference and therefore must have **in ref** specified.

- Operation Declaration Example

The following example shows the elements in an operation declaration:

```
type
    matrix = array[1..10] of int;

[broadcast] procedure array_$add ( in bind_var : handle_t;
                 in array1, array2 : matrix;
                 out result : integer);

[idempotent] function true_or_false ( in bind_var : handle_t;
                in high_val, low_val : real) : boolean;
```

Using NCS with FORTRAN Programs

The Network Computing System allows you to write distributed applications in FORTRAN. You define interfaces in the C syntax or the Pascal syntax of the Network Interface Definition Language (NIDL). The NIDL compiler generates C source code for the client stub and switch as well as for the server switch. These modules are compiled by a C compiler. Client and server application code can be written in FORTRAN, compiled by a FORTRAN compiler, and linked with the stubs and switches.

However, there are some special considerations that apply to FORTRAN, and there are additional considerations that apply to UNIX F77 FORTRAN. This section describes both general FORTRAN and F77 considerations.

General FORTRAN Considerations

This section describes considerations that apply to the use of any versions of FORTRAN with NCS. If you are building an application from source code both in FORTRAN and in C or Pascal, you should consult the appropriate language manuals for more information on cross-language communication.

Parameter Passing and Interface Definitions

FORTRAN passes all parameters by reference. Therefore, the NIDL declaration of any FORTRAN operation must specify that the stubs are to pass all parameters by reference. If you write the interface definition in the C syntax of NIDL, prefix all parameters with an * (asterisk), the indirection operator. If you use the Pascal syntax of NIDL, you must specify **in ref** for all input parameters. Special treatment of output parameters in the Pascal syntax is not required because the NIDL compiler assumes that the parameters are passed by reference.

Library Routine Considerations

FORTRAN programs cannot directly call library routines that expect any parameters to be passed by value. As a result, many **rpc_$** and **lb_$** routines cannot be called directly from FORTRAN. You must instead write C or Pascal interlude routines for the FORTRAN programs to call. These routines accept input parameters that are all passed by reference, make the calls passing parameters by value as required, and then return the results by reference.

The following NCS library routines cannot be called directly from FORTRAN:

- **rpc_$alloc_handle**
- **rpc_$bind**
- **rpc_$listen**
- **rpc_$name_to_sockaddr**
- **rpc_$set_binding**
- **rpc_$sockaddr_to_name**
- **rpc_$use_family**
- **rpc_$use_family_wk**
- **lb_$lookup_interface**
- **lb_$lookup_object**
- **lb_$lookup_object_local**
- **lb_$lookup_range**
- **lb_$lookup_type**
- **lb_$register**

NIDL Compiler F77 Options

The UNIX F77 compiler appends an _ (underscore) to external names. For example, if you specify a procedure name of **my_proc** in the source code, the compiler changes the name to **my_proc_**. While this renaming does not cause problems in programs that are generated entirely from FORTRAN source, it does require consideration when some of the program's modules are written in another language, since the C and Pascal compilers do not append underscores to names. The C or Pascal source code must explicitly terminate with an underscore any names that are referenced or defined by a FORTRAN program.

The NIDL compiler provides two options, **–f77c** and **–f77s**, that cause the compiler to generate stubs with names that conform to the F77 naming convention. As a result, you can write the interface definition without specifically appending underscores to any names. These options also make it easier to write an application in which the client is written in FORTRAN and the server is written in C or Pascal (or the opposite).

Use the **–f77c** option if the client is written in FORTRAN and is compiled by a UNIX F77 compiler. This option causes the NIDL compiler to use F77-compatible names (with underscores appended) for the client switch procedures. (The switch procedures are those actually invoked when a client makes a remote procedure call.)

Use the **–f77s** option if the server manager procedures (the procedures which actually implement the operations in an interface) are written in FORTRAN and are compiled by a UNIX F77 compiler. This option causes the NIDL compiler to generate server stub procedures that use F77-compatible names in calling the manager procedures.

The Location Broker (NCS)

This section describes the concepts behind the Location Broker and how it works. For information on how to configure the Location Broker on your network or internet, see Configuring NCS in *Communication Concepts and Procedures.*

Understanding the Location Broker

The Location Broker provides clients with information about the locations of objects and interfaces. Servers register with the Location Broker their socket addresses and the objects and interfaces to which they provide access. Clients issue requests to the Location Broker for the locations of objects and interfaces they wish to access. The broker returns database entries that match an object, type, interface, or combination of these, as specified in the request.

The Location Broker also implements the RPC message-forwarding mechanism. If a client sends a request for an interface to the Location Broker forwarding port on a host, the broker automatically forwards the request to the appropriate server on the host.

Location Broker Components

The Location Broker consists of the following interrelated components:

- **Local Location Broker** (LLB)

  An RPC server that maintains a database of information about objects and interfaces located on the local host. The LLB provides access to its database for application programs and also provides the Location Broker forwarding service. An LLB must run on any host that runs RPC servers. The LLB runs as the daemon program, **llbd**.

- **Global Location Broker** (GLB)

  An RPC server that maintains information about objects and interfaces throughout the network or internet. The GLB runs as the **nrglbd** daemon program.

- **Location Broker Client Agent**

  A set of library routines that application programs call to access LLB and GLB databases. Any client that uses Location Broker library routines is actually making calls to the Client Agent. The Client Agent interacts with LLBs and GLBs to provide access to their databases.

The following figure shows the relationships among application programs, the Location Broker components, and the Location Broker databases:

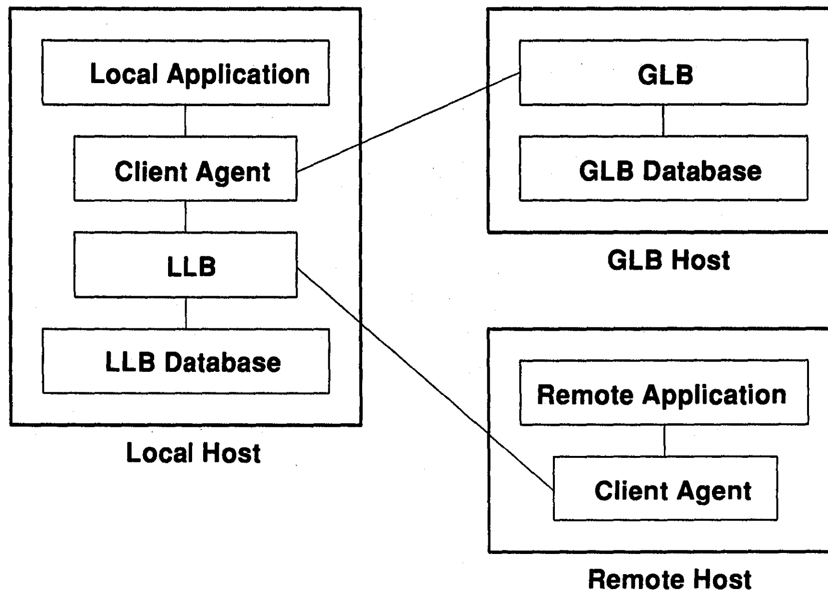


Figure 14. Location Broker Software

Location Broker Data

Each entry in a Location Broker database contains information about an object, an interface, and the location of a server that exports the interface to the object. The fields in a database entry are as follows:

Object UUID — The unique identifier of the object

Type UUID — The unique identifier that specifies the type of the object

Interface UUID — The unique identifier of the interface to the object

Flag — A flag that indicates if the object is global (and should be registered in the GLB database)

Annotation — 64 characters of user-defined information

Socket Address Length — The length of the socket address field

Socket Address — The location of the server that exports the interface to the object.

Each database entry contains one object UUID, one interface UUID, and one socket address. This means a Location Broker database must have an entry for each possible combination of object, interface, and socket address. For example, the database must have ten entries for a server that does all of the following:

- Listens on two sockets, **socket_a** and **socket_b**.
- Exports **interface_1** for **object_x**, **object_y**, and **object_z**.
- Exports **interface_2** for **object_p** and **object_q**.

The server must make a total of ten calls to the **lb_$register** routine to completely register its interfaces and objects.

You can look up Location Broker information by using any combination of the object UUID, type UUID, and interface UUID as keys. You can also request the information from the GLB database or from a particular LLB database. Therefore, you can obtain information about all objects of a specific type, all hosts with a specific interface to a object, or even all objects and interfaces at a specific host. For example, you can find the addresses of all remotely available array processors by looking up all entries with the **arrayproc** type.

The Location Broker Client Agent

The Location Broker Client Agent is a set of library routines that applications use to access and modify the LLB and GLB databases. When a program issues any Location Broker call, the call actually goes to the local host's Client Agent. The Client Agent then does the work to add, delete, or look up information in the appropriate Location Broker database.

The following figure illustrates a typical case in which a client requires a particular interface to a particular object, but does not know the location of a server exporting the interface to the object. In this figure, an RPC server registers itself with the Location Broker by calling the Client Agent in its host (step 1a). The Client Agent, through the LLB, adds the registration information to the LLB database at the server host (not shown). The Client Agent also sends the information to the GLB (1b). To locate the server, the client issues a Location Broker lookup call (2a). The Client Agent on the client host sends the lookup request to the GLB, which returns it through the Client Agent to the client (2b). The client can then use RPC calls to communicate directly with the located server (3a, 3b).

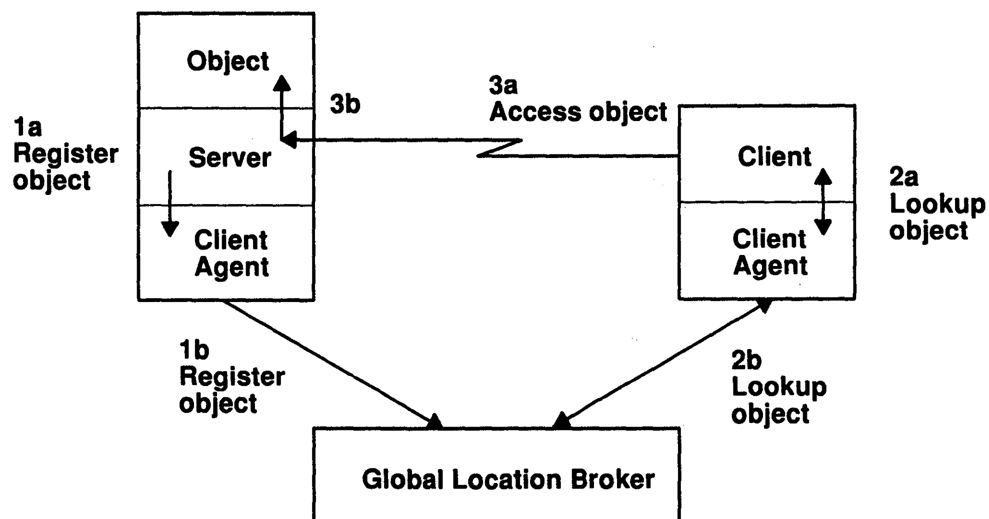


Figure 15. Client Agent and a Global Location Broker

If a client knows the host where the object is located without knowing the port number used by the server, it can specify this information in its lookup call. Then the Client Agent interrogates the remote host's LLB directly, as illustrated in the following figure.

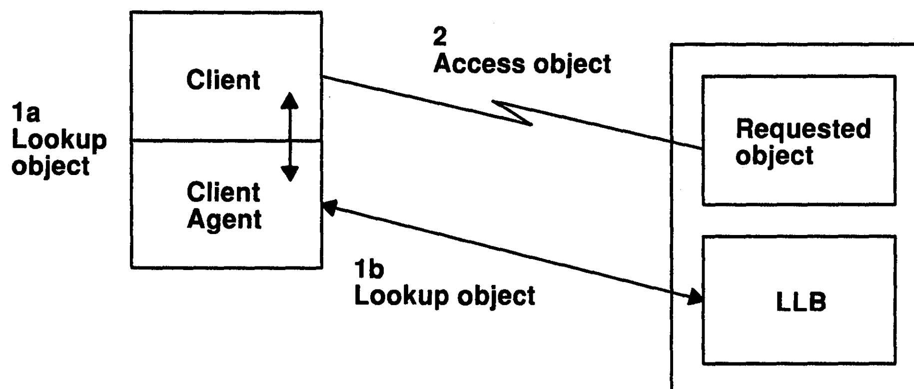


Figure 16. Client Agent Doing a Lookup at a Known Host

The Local Location Broker (LLB)

The LLB, which runs as the **llbd** daemon, maintains a database of the objects and interfaces that are exported by servers running on the host. In addition, it acts as a forwarding agent for requests.

An **llbd** daemon must be running on hosts that run RPC servers. However, it is recommended to run an **llbd** daemon on every host in the network or internet.

The Local Database

The database maintained by the LLB provides location information about interfaces on the local host. This information is used by both local and remote applications. To look up information in an LLB database, an application queries the LLB through a Client Agent. For applications on a local host, the Client Agent accesses the LLB database directly. For applications on a remote host, the remote Client Agent accesses the LLB database through the LLB process. You can also access the LLB database manually by using the **lb_admin** command.

The LLB Forwarding Agent

The LLB's forwarding facility eliminates the need for a client to know the specific port that a server uses. It is intended to limit the number of well-known port numbers that must be reserved for specific purposes.

The forwarding agent listens on one well-known port for each address family. It forwards any messages that it receives to the local server that exports the requested object. Forwarding is particularly useful when the requester of a service already knows the host where the server is running. For example, you would not need to assign a well-known port to a server that reports load statistics, nor would you need to register the server with the GLB. Each such server would register only with its host's LLB. Remote clients would access the server by specifying the object, the interface, and the host, but not a specific port, when making a remote procedure call.

The Global Location Broker (GLB)

The GLB, which runs as the **nrglbd** daemon, manages information about the objects and interfaces that are available to users on the network. In an internet, at least one GLB must be running on each network.

The GLB database is accessed manually by using the **lb_admin** command. The **lb_admin** command is useful to manually correct errors in the database. For example, if a server starts while the GLB is not running, you can manually enter the information for the server in the GLB database. Similarly, if a server terminates abnormally without unregistering itself, you can use the **lb_admin** command to manually remove its entry from the GLB database.

NCS Daemons and Utilities

The description of each NCS daemon and utility program includes the following information:

- A **Purpose** section identifying the use of the daemon or utility
- A **Syntax** section showing the syntax or location of the program
- A **Description** section giving an explanation of the usage
- A **Files** section describing any files related to the operation of the program.

Some of the daemons and utilities also include descriptions of subcommands that are used along with the program, and examples of command usage.

List of Daemons and Utilities

The following is a list of NCS-related commands and daemons. The descriptions of each can be found in *Commands Reference*.

- **lb_admin** command
- **llbd** daemon
- **nidl** command
- **nrglbd** daemon
- **uuid_gen** command.

rpc_$ Library Routines (NCS)

This section provides reference descriptions of the **rpc_$** library routines. These routines implement the NCS Remote Procedure Call (RPC) mechanism.

The definitions of the constants and types associated with the **rpc_$** routines and descriptions of error values that are specific to each routine are discussed in the following. Error values returned by other subsystem procedures are not documented.

Note: In these routines, all input parameters except integers and pointers are passed by reference, and all output parameters are passed by reference.

Constants

The following constants are used in **rpc_$** routines. The value of the constant is enclosed in ( ) parentheses.

socket_$unspec_port (0)
: Port number indicating to the RPC runtime software that no port is specified.

The following 16-bit integer constants are used in specifying the communications protocol address families in **socket_$addr_t** structures:

socket_$unspec (0)
: Address family is unspecified.

socket_$internet (2)
: Internet Protocols (IP).

socket_$dds (13)
: Domain protocol (DDS).

Note: The **rpc_$use_family** and **rpc_$use_family_wk** routines use the 32-bit integer equivalents of these values.

External Variable

The following external variable is used in **rpc_$** routines:

uuid_$nil
: An external **uuid_$t** variable that is preassigned the value of the nil UUID. Do not change the value of this variable.

Data Types

The following data types are used in **rpc_$** routines:

handle_t
: An RPC handle.

rpc_$epv_t
: An entry point vector (EPV), which specifies an array of server stub procedures.

rpc_$if_spec_t
: An RPC interface specifier. This opaque data type contains information about an interface, including its UUID, the current version number, any well-known ports used by servers that export the interface, and the number of operations in the interface.

rpc_$server_stub_t
: A pointer to a server stub procedure. This type is used in the server stub EPV to point to the stub representation of a called procedure.

socket_$addr_t

A socket address record that uniquely identifies a socket. A record with the following fields:

Family	16-bit integer	The address family
Socket Address	14 bytes	The socket address.

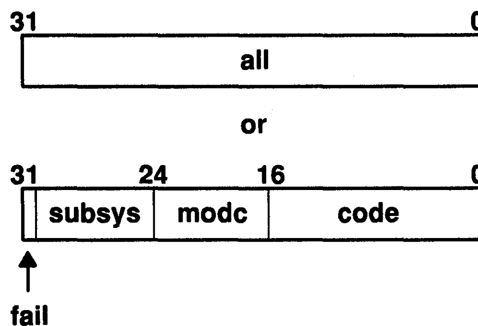
The format of the address depends on the address family.

status_$t

A status code. The following C **typedef** statement defines the **status_$t** data type:

```
typedef union {
    struct {
        unsigned fail : 1,
               subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_$t;
```

The following illustrates this type:



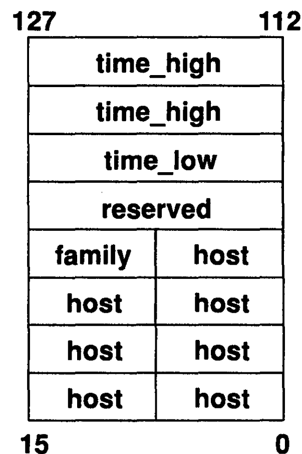
Field	Description
all	All 32 bits in the status code.
fail	The fail bit. If this bit is set, the error was not within the scope of the module invoked, but in a lower level module.
subsys	The subsystem that encountered the error.
modc	The module that encountered the error.
code	A signed number that identifies the type of error that occurred.

uuid_$t

A 128-bit value that uniquely identifies an object, type, or interface for all time. The following C **typedef** statement defines the **uuid_$t** data type:

```
typedef struct uuid_$t {
    unsigned long time_high;
    unsigned short time_low;
    unsigned short reserved;
    unsigned char family;
    unsigned char (host)[7];
} uuid_$t;
```

The following illustrates this type:



Field	Description
time_high	The high 32 bits of a 48-bit unsigned time value, which is the number of 4-microsecond intervals that have passed between 1/1/1980 00:00 GMT and the time of UUID creation.
time_low	The low 16 bits of the 48-bit time value.
reserved	16 bits of reserved space.
family	An 8–bit address family identifier corresponding to the value used in a **socket_$addr_t** record.
host	7 bytes that specify the host on which the UUID was created. The format depends on the address family of the host.

rpc_$ Status Codes

The following status codes for errors are returned by RPC routines:

rpc_$comm_failure The client is unable to get a response from the server.

rpc_$op_rng_error The requested operation does not correspond to a valid operation in the requested interface.

rpc_$unk_if The requested interface is not known. It is not registered in the server, the version number of the registered interface is different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.

rpc_$cant_create_sock The RPC runtime library is unable to create a socket.

rpc_$cant_bind_sock The RPC runtime library created a socket but was unable to bind it to a socket address.

rpc_$wrong_boot_time The server boot time value maintained by the client does not correspond to the current server boot time. (The server was probably rebooted while the client program was running.)

rpc_$too_many_ifs The maximum number of interfaces is already registered with the RPC runtime library. The server must unregister some interface before it registers an additional interface.

rpc_$not_in_call An internal error.

rpc_$you_crashed This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, and then receives a response after the server restarts.

rpc_$proto_error An internal protocol error.

rpc_$too_many_sockets The server is trying to use more than the maximum number of sockets allowed. The server has called the **rpc_$use_family** or **rpc_$use_family_wk** routine too many times.

rpc_$illegal_register You are trying to register an interface that is already registered and you are using an EPV different from the one used when the interface was first registered. An interface can be multiply registered, but you must use the same EPV in each call to the **rpc_$register** routine.

rpc_$bad_pkt The server or client has received an ill-formed packet.

rpc_$unbound_handle The handle is not bound and does not represent a particular host address. This status code is returned by the **rpc_$inq_binding** routine.

rpc_$addr_in_use The address and port specified in a call to the **rpc_$use_family_wk** routine are already in use. This is caused by multiple calls to the **rpc_$use_family_wk** routine with the same well-known port.

List of rpc_$ Library Routines

The following routines are found in *Calls and Subroutines Reference.*

- **rpc_$alloc_handle**
- **rpc_$bind**
- **rpc_$clear_binding**
- **rpc_$clear_server_binding**
- **rpc_$dup_handle**
- **rpc_$free_handle**
- **rpc_$inq_binding**
- **rpc_$inq_object**
- **rpc_$listen**
- **rpc_$name_to_sockaddr**
- **rpc_$register**
- **rpc_$set_binding**
- **rpc_$sockaddr_to_name**
- **rpc_$unregister**
- **rpc_$use_family**
- **rpc_$use_family_wk**

pfm_$ Library Routines (NCS)

This section provides reference descriptions of the **pfm_$** library routines. These routines allow programs to manage signals, faults, and exceptions by establishing cleanup handlers.

Definitions of the constants and types associated with the **pfm_$** routines and descriptions of error values that are specific to the routines are discussed in the first section. Error values returned by other subsystem procedures are not documented. The *cleanup handler* is also discussed.

Note: In these routines, all input parameters except integers and pointers are passed by reference, and all output parameters are passed by reference.

Cleanup Handlers

A cleanup handler is a piece of code that ensures a program terminates gracefully when it receives a fatal error. A cleanup handler begins with a call to the **pfm_$cleanup** routine. It usually ends with a call to the **pfm_$signal** or **pfm_$exit** routine. It can also simply continue back into the program after the cleanup code.

A cleanup handler is not entered until all fault handlers established for a fault have returned. If there is more than one established cleanup handler for a program, the most recently established cleanup handler is entered first, followed by the next most recently established cleanup handler, and so on, up to the first established cleanup handler if necessary.

A default cleanup handler is invoked after all user-defined handlers have completed. The default cleanup handler releases any resources still held by the program, before returning control to the process that invoked it.

Constant

The following constant is used in **pfm_$** routines:

pfm_$init_signal_handlers Used as the flags parameter to the **pfm_$init** routine, causing C signals to be intercepted and converted to program fault management (PFM) signals.

Data Types

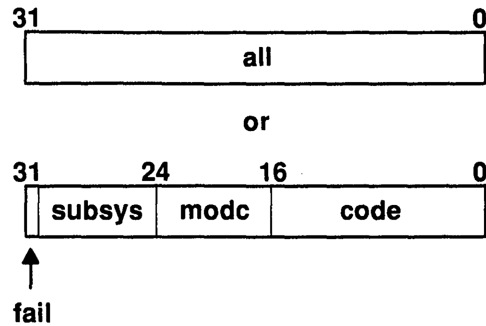
The following data types are used in **pfm_$** routines:

pfm_$cleanup_rec A record type for passing process context among cleanup handler routines. It is an opaque data type.

status_$t A status code. The following C **typedef** statement defines the **status_$t** data type:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_$t;
```

The following illustrates this type:



Field	Description
all	All 32 bits in the status code.
fail	The fail bit. If this bit is set, the error was not within the scope of the module invoked, but in a lower level module.
subsys	The subsystem that encountered the error.
modc	The module that encountered the error.
code	A signed number that identifies the type of error that occurred.

pfm_$ Status Codes

The following status codes for errors are returned by program fault management routines:

pfm_$bad_rls_order	An attempt was made to release a cleanup handler out of order.
pfm_$cleanup_not_found	There is no pending cleanup handler.
pfm_$cleanup_set	A cleanup handler was established successfully.
pfm_$cleanup_set_signalled	An attempt was made to use **pfm_$cleanup_set** as a signal.
pfm_$invalid_cleanup_rec	An invalid cleanup record was passed to a routine.
pfm_$no_space	Storage cannot be allocated for a cleanup handler.
status_$ok	The routine was successful.

List of pfm_$ Library Routines

The following routines are found in *Calls and Subroutines Reference*

- **pfm_$cleanup**
- **pfm_$enable**
- **pfm_$enable_faults**
- **pfm_$inhibit**
- **pfm_$inhibit_faults**
- **pfm_$init**
- **pfm_$reset_cleanup**
- **pfm_$rls_cleanup**
- **pfm_$signal**

lb_$ Library Routines (NCS)

This section provides reference descriptions of the **lb_$** library routines. These constitute the programming interface to the Location Broker.

Definitions of the constants and types associated with the **lb_$** routines and descriptions of error values that are specific to the routines are discussed in the first section. Error values returned by other subsystem procedures are not documented.

Note: In these routines, all input parameters except integers and pointers are passed by reference, and all output parameters are passed by reference.

Constants

The following constants are used in **lb_$** routines:

lb_$default_lookup_handle
: Assigned to an **lb_$lookup_handle_t** variable for use as an input parameter value. If you use this constant, a Location Broker lookup routine will start searching at the beginning of the database. The value of this constant is 0 (zero).

lb_$server_flag_local
: Used in the **flags** field of an **lb_$entry_t** variable. This constant specifies that the server that implements the interface is available to the local node only. It is registered only in the Local Location Broker (LLB) database. If this flag is not set, the information for the entry is also registered in the Global Location Broker (GLB) database. The value of this constant is 1 (one).

External Variable

The following external variable is used in **lb_$** lookup routines:

uuid_$nil
: An external **uuid_$t** variable that is preassigned the value of the nil Universal Unique Identifier (UUID). It is used as a wild card in Location Broker lookup operations. Do not change the value of this variable.

Data Types

The following data types are used in **lb_$** routines:

lb_$entry_t

An identifier for an object, a type, an interface, and the socket address used to access a server exporting the interface to the object. The record contains the following fields:

Field	Type	Description
object	**uuid_$t**	The UUID for the object.
obj_type	**uuid_$t**	The UUID for the type of the object.
obj_interface	**uuid_$t**	The UUID for the interface.
flags	**lb_$server_flag**	Must be 0 or **lb_$server_flag_local**.
annotation	64-character array	User-defined textual annotation.
saddr_len	32-bit integer	The length of the socket address (**saddr**) field.
saddr	**socket_$addr_t**	The socket address of the server.

Note: If the object, type, or interface is not associated, the value for the UUID can be **uuid_$nil**.

lb_$lookup_handle_t

A 32-bit integer used to specify the location in the database at which a Location Broker lookup operation starts.

socket_$addr_t

A socket address record that uniquely identifies a socket. The record contains the following fields:

Family	16-bit integer	The address family
Socket Address	14 bytes	The socket address.

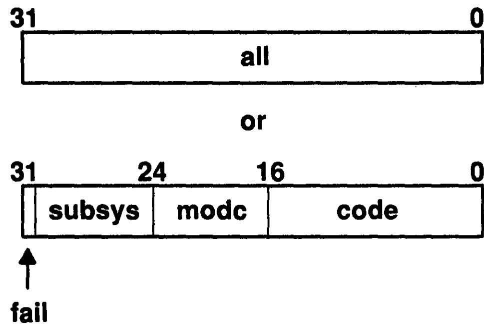
The format of the address depends on the address family.

status_$t

A status code. The following C **typedef** statement defines the **status_$t** data type:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_$t;
```

The following illustrates this type:



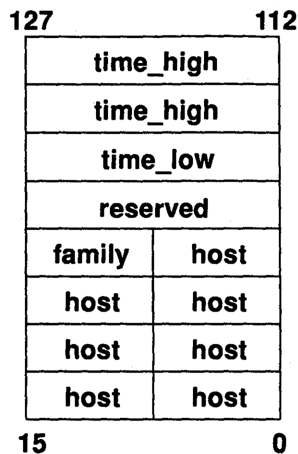
Field	Description
all	All 32 bits in the status code.
fail	The fail bit. If this bit is set, the error was not within the scope of the module invoked, but in a lower level module.
subsys	The subsystem that encountered the error.
modc	The module that encountered the error.
code	A signed number that identifies the type of error that occurred.

uuid_$t

A 128-bit value that uniquely identifies an object, type, or interface for all time. The following C **typedef** statement defines the **uuid_$t** data type:

```
typedef struct uuid_$t {
    unsigned long time_high;
    unsigned short time_low;
    unsigned short reserved;
    unsigned char family;
    unsigned char (host)[7];
} uuid_$t;
```

The following illustrates this type:



Field	Description
time_high	The high 32 bits of a 48-bit unsigned time value, which is the number of 4-microsecond intervals that have passed between 1/1/1980 00:00 GMT and the time of UUID creation.
time_low	The lower 16 bits of the 48-bit time value.
reserved	16 bits of reserved space.
family	An 8-bit address family identifier corresponding to the value used in a **socket_$addr_t** record.
host	7 bytes that specify the host on which the UUID was created. The format depends on the address family of the host.

lb_$ Status Codes

The following status codes for errors are returned by the Location Broker routines:

lb_$database_invalid The format of the Location Broker database is out of date for one of the following reasons:

- The database may have been created by an old version of the Location Broker. Delete the out-of-date database and register again any entries that it contained.
- The LLB or GLB database being accessed may be running out-of-date software. Update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered The Location Broker does not have any entries that match the criteria specified in the lookup or unregister call. The requested object, type, interface, or combination of these items is not registered in the specified database. If you are calling an **lb_$lookup_object_local** or **lb_$lookup_range** routine specifying an LLB, check that you have specified the correct LLB database.

lb_$update_failed The Location Broker is unable to register or unregister the entry (for example, because the broker ran out of disk space).

lb_$cant_access The Location Broker cannot access the database for one of the following reasons:

The database does not exist, and the Location Broker cannot create it.

The database exists, but the Location Broker cannot access it.

The GLB entry table or the GLB propagation queue is full.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested GLB or LLB database. A communications failure has occurred or the broker was not running.

List of lb_$ Library Routines

The following is a list of NCS-related routines. Descriptions of each can be found in *Calls and Subroutines Reference*.

- **lb_$lookup_interface**
- **lb_$lookup_object**
- **lb_$lookup_object_local**
- **lb_$lookup_range**
- **lb_$lookup_type**
- **lb_$register**
- **lb_$unregister**

uuid_$ Library Routines (NCS)

This section provides reference descriptions of the **uuid_$** library routines. These create and access Universal Unique Identifiers (UUIDs).

Definitions of the external variables and data types associated with the **uuid_$** routines are discussed first.

Note: In these routines, all input parameters, except integers and pointers, are passed by reference. All output parameters are passed by reference as well.

External Variables

The following external variables are used in **uuid_$** routines:

uuid_$nil
An external **uuid_$t** variable that is preassigned the value of the nil UUID. Do not change the value of this variable.

uid_$nil
An external **uid_$t** variable that is preassigned the value of the nil UID. Do not change the value of this variable.

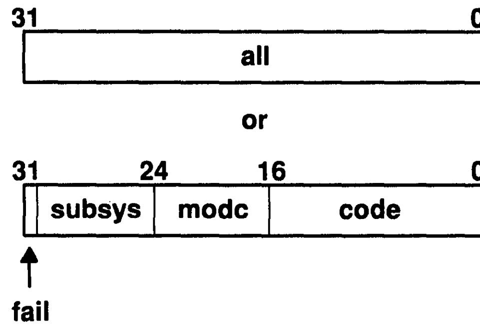
Data Types

The following data types are used in **uuid_$** routines:

status_$t
A status code. The following C **typedef** statement defines the **status_$t** data type:

```
typedef union {
   struct {
        unsigned fail : 1,
                     subsys : 7,
                   modc : 8;
        short    code;
     } s;
    long all;
 } status_$t;
```

The following illustrates this type.



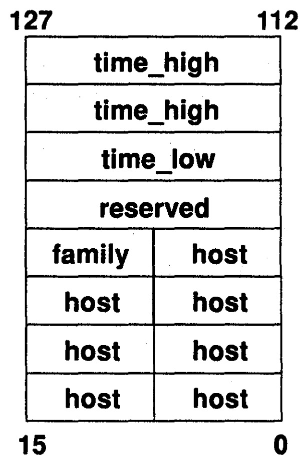
Field	Description
all	All 32 bits in the status code.
fail	The fail bit. If this bit is set, the error was not within the scope of the module invoked, but in a lower level module.
subsys	The subsystem that encountered the error.
modc	The module that encountered the error.
code	A signed number that identifies the type of error that occurred.

uid_$t A 64-bit value that uniquely identifies objects and types.

uuid_$t A 128-bit value that uniquely identifies an object, type, or interface for all time. The following C **typedef** statement defines the **uuid_$t** data type:

```
typedef struct uuid_$t {
     unsigned long time_high;
     unsigned short time_low;
     unsigned short reserved;
     unsigned char family;
     unsigned char (host)[7];
 } uuid_$t;
```

The following illustrates this type:



- Field Description

time_high The high 32 bits of a 48-bit unsigned time value, which is the number of 4-microsecond intervals that have passed between 1/1/1980 00:00 GMT and the time of UUID creation.

time_low The lower 16 bits of the 48-bit time value.

reserved 16 bits of reserved space.

family An 8-bit address family identifier corresponding to the value used in a **socket_$addr_t**.

host 7 bytes that specify the host on which the UUID was created. The format depends on the address family of the host.

uuid_$string_t A string of 37 characters (including a null terminator) that is an ASCII representation of a UUID. It takes the following form:

cccccccccccc.ff.h1.h2.h3.h4.h5.h6.h7

The *cccccccccccc* field specifies the time stamp in hexadecimal digits. The *ff* field specifies the address family, in hexadecimal digits, and *h1 ... h7* specifies the 7-byte host ID, in hexadecimal digits.

List of uuid_$ Library Routines

The following is a list of NCS-related routines. Descriptions of each can be found in *Calls and Subroutines Reference*.

- **uuid_$decode**
- **uuid_$encode**
- **uuid_$gen**

Glossary

address family. A set of communications protocols that use a common addressing mechanism to identify end points. This term is often used synonymously with *protocol family.*

allocate (a handle). To create a Remote Procedure Call (RPC) handle that identifies an object.

bind. To set a binding. NCS provides two library routines that bind: **rpc_$bind**, which both creates and binds a handle, and **rpc_$set_binding**, which requires a handle as an input parameter.

binding. A temporary association between a client and both an object and a server that exports an interface to the object. A binding is meaningful only to the program that sets it and is represented by a bound handle.

broker. A server that manages information about objects and interfaces to the objects. A program that wishes to become the client of an interface can use a broker to obtain information about servers that export the interface. Location brokers are brokers.

client. A user of an interface. In the context of this manual, a program that makes remote procedure calls.

Client Agent. See *Location Broker Client Agent.*

entry point vector (EPV). A record whose fields are pointers to procedures that implement the operations defined by an interface.

EPV. See *entry point vector.*

export. To provide the operations defined by an interface. A server exports an interface to a client. See also *import.*

GLB. See *Global Location Broker.*

Global Location Broker (GLB). Part of the NCS Location Broker. A server that maintains global information about objects on a network or an internet.

handle. A data structure that is a temporary local identifier for an object. You create a handle by allocating it. You make a handle identify an object at a specific location by binding it.

host. A computer that is attached to a network.

host ID. An identifier for a host. A host ID uniquely identifies a host within an address family on a network, but does not identify the network. A host ID is not necessarily sufficient to establish communications with a host. See also *network address.*

idempotent. A class of operations. An operation is idempotent if its results do not affect the results of any operation. For example, a call that returns the time is idempotent.

import. To request the operations defined by an interface. A client imports an interface from a server. See also *export.*

interface. A set of operations. The Network Computing Architecture specifies a Network Interface Definition Language for defining interfaces.

Internet Protocol (IP). The protocol that provides the interface from the higher level host-to-host protocols to the local network protocols. Addressing at this level is usually from host to host.

IP. See *Internet Protocol.*

LLB. See *Local Location Broker.*

Local Location Broker (LLB). Part of the NCS Location Broker. A server that maintains information about objects on the local host. The LLB also provides the Location Broker forwarding facility.

Location Broker. A set of software including the Local Location Broker, the Global Location Broker, and the Location Broker Client Agent. The Location Broker maintains information about the locations of objects.

Location Broker Client Agent. Part of the NCS Location Broker. Programs communicate with Global Location Brokers and with remote Local Location Brokers using the Location Broker Client Agent.

marshall. To copy data into a Remote Procedure Call (RPC) packet. Stubs perform marshalling. See also *unmarshall.*

NCK. See *Network Computing Kernel.*

NCS. See *Network Computing System.*

network address. A unique identifier (within an address family) for a specific host on a network or an internet. The network address is sufficient to identify a host, but does not identify a communications end point within the host.

Network Computing Architecture. A set of protocols and architectures that support distributed computing.

Network Computing Kernel (NCK). The combination of the RPC runtime library and the Location Broker, which contain the necessary pieces required to run distributed applications.

Network Computing System (NCS). A set of software tools developed by Apollo Computer Inc. that conform to the Network Computing Architecture. These tools include the Remote Procedure Call runtime library, the Location Broker, and the NIDL compiler.

Network Interface Definition Language (NIDL). A declarative language for the definition of interfaces. A component of the Network Computing Architecture. NIDL has two forms, a Pascal-like syntax and a C-like syntax.

NIDL. See *Network Interface Definition Language.*

NIDL compiler. An NCS tool that converts an interface definition, written in NIDL, into several program modules, including source code for client and server stubs. The NIDL compiler accepts interface definitions written in either syntax of NIDL. It generates C source code and C or Pascal header files.

object. An entity that is manipulated by well-defined operations. Disk files, printers, and array processors are examples of objects. Objects are accessed though interfaces. Every object has a type.

object UUID. A UUID that identifies a particular object. Both the RPC runtime library and the Location Broker use object UUIDs to identify objects.

operation. A procedure through which an object is accessed or manipulated. An operation is defined syntactically by its name and its parameters but not by its implementation.

PFM. See *program fault management.*

port. A specific communications end point within a host. A port is identified by a port number. See also *socket.*

program fault management (PFM). A subsystem of NCS that allows a user to set up cleanup routines when an application fails to complete successfully.

protocol family. A set of related communications protocols, for example, the Department of Defense Internet Protocols. All members of a protocol family use a common addressing mechanism to identify end points. This term is often used synonymously with *address family.*

register (an interface). To make an interface known to the RPC runtime library, and thereby available to clients through the RPC mechanism. The **rpc_$register** routine registers an interface.

register (an object). To enter an object and its location in the Location Broker database. The **lb_$register** routine registers an object with the Location Broker. A program can use Location Broker lookup routines to determine the location of a registered object.

remote procedure call. An invocation of a remote operation. You can make remote procedure calls between processes on different hosts or on the same host.

Remote Procedure Call runtime library. The set of **rpc_$** library routines that NCS provides to implement a remote procedure call mechanism.

RPC. See *remote procedure call.*

server. A process that implements interfaces. In the context of this manual, a server whose procedures can be invoked from remote hosts. A server exports one or more interfaces to one or more objects.

set (a binding). To associate an allocated Remote Procedure Call (RPC) handle with a specific socket address.

socket. A port on a specific host; a communications end point that is accessible through a protocol family's addressing mechanism. A socket is identified by a socket address.

socket address. A data structure that uniquely identifies a specific communications end point. A socket address consists of a port number and a network address. It also specifies the address family (protocol family).

stub. A program module that transfers remote procedure calls and responses between a client and a server. Stubs perform marshalling, unmarshalling, and data format conversion. Both clients and servers have stubs. The NIDL compiler generates client and server stub code from an interface definition.

type. A class of object. All objects of a specific type can be accessed though the same interface or interfaces.

type UUID. A UUID that permanently identifies a particular type. Both the RPC runtime library and the Location Broker use type UUIDs to specify types.

UID. See *user number.*

Universal Unique Identifier (UUID). A 128-bit value used for identification. NCS uses UUIDs to identify interfaces, objects, and types.

unmarshall. To copy data from an RPC packet. Stubs perform unmarshalling. See also *marshall.*

user number. A number that uniquely identifies a user to a system.

UUID. See *Universal Unique Identifier.*

Chapter 6. Network Management/6000 (xgmon)

The AIX Network Management/6000 Licensed Program (**xgmon**) is a network management program for monitoring TCP/IP networks. It assists in monitoring the status of all the machines on a network by communicating with SNMP agents and receiving SNMP–based traps. This chapter provides detailed information on the **xgmon** programming utility, which enables you to extend the **xgmon** program, as well as information on the SNMP API Subroutine Library, which allows you to create SNMP manager applications. Also included is information on the Simple Network Management Protocol (SNMP), the SNMP daemon, and the SNMP Command Line Manager.

xgmon Overview

The **xgmon** network monitor program is designed so that it can be readily customized by the end user. However, the end user must know where the relevant directories are and which features can be altered by setting the environment variables appropriately. The relevant directories are:

- **/usr/lpp/xgmon**
- **/usr/lpp/xgmon/bin**
- **/usr/lpp/xgmon/lib**
- **/etc**
- **$HOME**

The **xgmon** client (manager) is a monitor for TCP/IP networks. To understand the **xgmon** program and exploit its full potential, the programmer needs to have a working knowledge of AIX, C programming, and general TCP/IP network management protocols. The following RFCs will also be of use:

- RFC 1098, Simple Network Management Protocol (SNMP)
- RFC 1066, Management Information Base for Network Management of TCP/IP-based internets (MIB)
- RFC 1065, Structure and Identification of Management Information for TCP/IP-based internets (SMI).

Consult the **xgmon** Overview for Network Management in *Communication Concepts and Procedures* for more information on xgmon.

Environment variables, or shell variables, are set by the user to define the current shell environment. Environment variables that are used by the **xgmon** program are:

GLIB Defines the path for the **xgmon** library programs. The **xgmon** program uses the default setting of the **/usr/lpp/xgmon/lib** directory for this variable.

Note: If the network manager chooses to use a different library program directory, the **GLIB** environment can be set to this other directory *before* the **xgmon** program is invoked.

XGMONFONT Defines the X11 font used in the topology display window and the virtual G machine windows. See the /**usr/lpp/fonts** directory for the available X11 fonts. The **xgmon** program uses the default setting of 8x13 for this variable.

Note: The font specified should be of fixed width for proper orientation. The **xgmon** client (manager) deals correctly with variable width fonts in a topology display window, but the results will not be as desired when used in a virtual G machine output window.

Set and export environment variables as follows:

EnvironmentVariable= *Value*
export *EnvironmentVariable*

As shipped, the **xgmon** distribution resides under the /**usr/lpp/xgmon** directory. The **xgmon** binary is found under the **bin** subdirectory, and the library commands are stored under the **lib** subdirectory. This binary may be copied or moved to a directory already in the user's search path (such as /**usr/bin**), or the directory itself may be added to the search path.

The **xgmon** program must be run with root privileges to support all of its capabilities. The **xgmon** program is therefore installed as a *setuid* program owned by the root user.

The **xgmon** client (manager) obtains its description of Management Information Base (MIB) variables (described by RFC 1066) from the **mib_desc** file found in the /**etc** directory. For more information on these variables, see Working with Management Information Base (MIB) Variables on page 6–9. For more information on the tree structure of the MIB database, see Understanding the Management Information Base (MIB) on page 6–3.

The **xgmon** program can be extended by a C programmer. This is done by making new intrinsic functions available to library programs. A user-written intrinsic function can override an intrinsic function already in the **xgmon** program. This is done by giving the new intrinsic function the same name as the existing function it replaces. See How to Create **xgmon** Intrinsic Functions on page 6–43 and Extending **xgmon** Intrinsic Functions on page 6–23.

The **xgmon** program contains a compiler that translates programs written in the **xgmon** programming utility into object code that is interpreted by virtual G machines (VGMs). The operation of the VGMs is overseen by the **xgmon** program. Each VGM works on its own library program. More than one library program can execute at the same time because the **xgmon** client (manager) is able to control more than one VGM at a time. See Working with the Virtual G Machine (VGM) Output Windows in *Communication Concepts and Procedures*.

The **xgmon** program is able to communicate with SNMP agents. It is also able to receive SNMP-based traps. Trap information may arrive by way of the conventional SNMP UDP packets. These traps are made available to library programs executing within virtual G machines. In addition, the **ping** command (ICMP ECHO/ECHO RESPONSE) can be used within the **xgmon** program to check the reachability of hosts.

The **xgmon** program supports the following hardware:

- IBM RISC System/6000
- IBM Megapel display (5081)
- IBM ASCII terminals (line mode).

The **xgmon** client (manager) supports the following software platform:

- AIX 3.1.

In principle, the code should run on any AIX system, but only the above system has been tested.

Understanding the Simple Network Management Protocol (SNMP)

The Simple Network Management Protocol (SNMP) is a protocol used by network hosts to exchange information used in the management of networks. SNMP is defined in Requests For Comments (RFCs), available from the Network Information Center at SRI International, Menlo Park, California.

The following RFCs define SNMP:

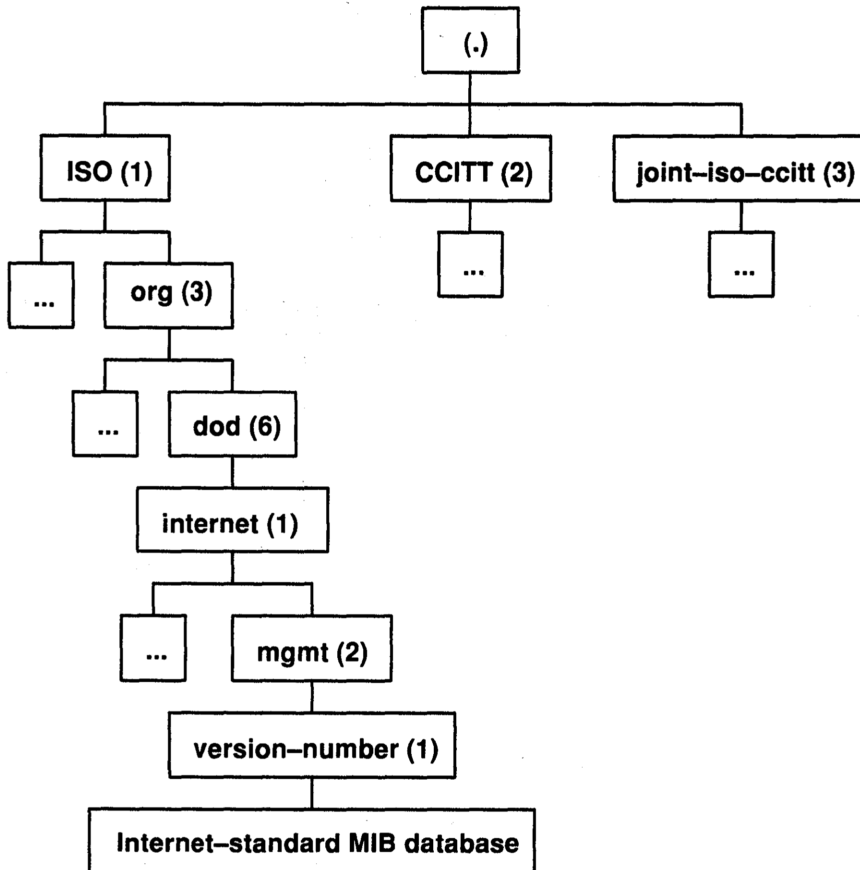
RFC1065	Defines the structure of Management Information.
RFC1066	Defines the Management Information Base (MIB) for network management.
RFC1098	Defines the SNMP protocol for creating requests for MIB information and formatting responses.

SNMP network management is based on the familiar client-server model that is widely used in TCP/IP-based network applications. Each host that is to be managed runs a process called an agent. The agent is a server process that maintains the MIB database for the host. Hosts that are involved in network management decision-making may run a process called a monitor. A monitor is a client application that generates requests for MIB information and processes responses. In addition, a monitor may send requests to agent servers to modify MIB information.

Understanding the Management Information Base (MIB)

The Management Information Base (MIB) is a database containing the information pertinent to network management. The database is conceptually organized as a tree. The upper structure of this tree is defined in RFC 1065 and RFC 1066. The internal nodes of the tree represent subdivision by organization or function. MIB variable values are stored in the leaves of this tree. Thus, every distinct variable value corresponds to a unique path from the root of the tree. The children of a node are numbered sequentially from left to right, starting from 1, so that every node in the tree has a unique name, which consists of the sequence of node numbers that comprise the path from the root of the tree to the node.

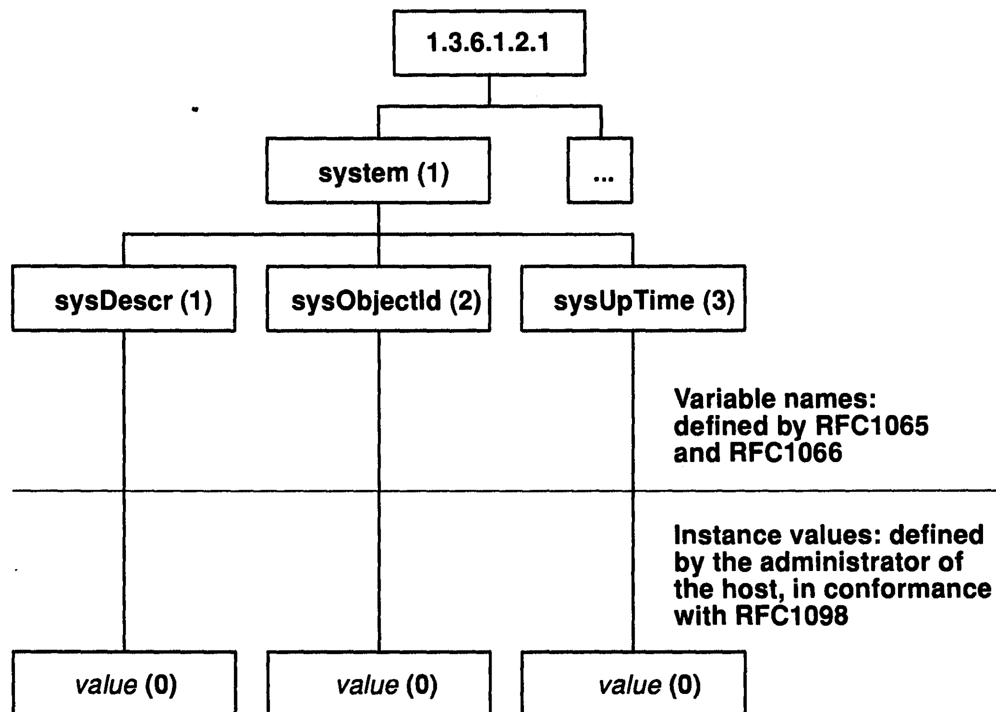
The following figure is a representation of part of the MIB tree:



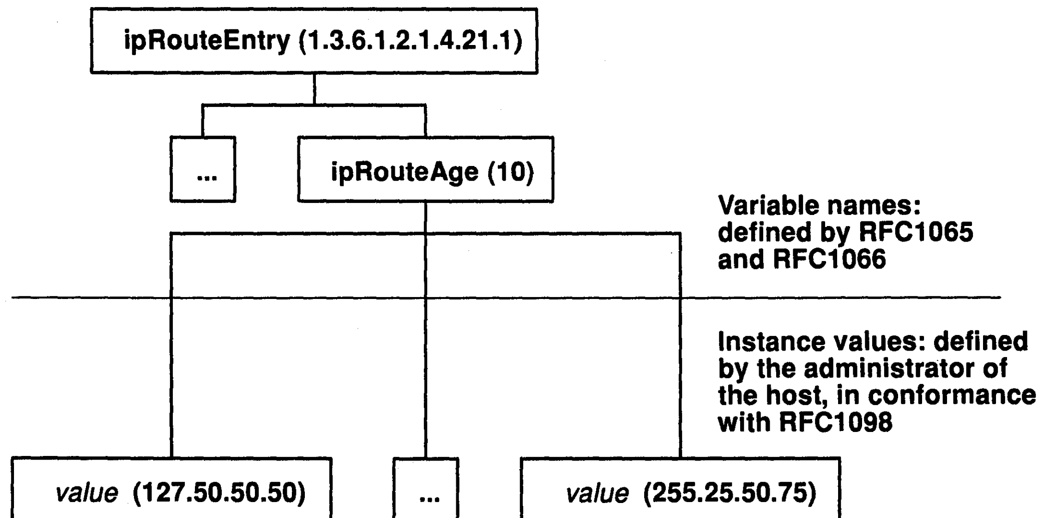
The network management data for the Internet is stored in the subtree reached by the path 1.3.6.1.2.1. This notation is the conventional way of writing the numeric path name, separating node numbers by periods. All variables defined in RFC 1066 have numeric names that begin with this prefix.

Note: Future versions of the Internet-standard MIB may have higher version numbers. The names of variables will therefore be distinct from those of earlier versions.

A typical variable value is stored as a leaf, as represented in the following figure:



The values of the variables are data associated by the MIB manager with each uniquely named instance of a variable. For example, 1.3.6.1.2.1.1.0 is the unique name of the system description, a text string describing the host's operational environment. There is only one such string, so the instance of the variable name 1.3.6.1.2.1.1 is denoted by a 0 (zero), which is reserved for this use only. Many other variables have multiple instances as shown in the following figure:



Each variable that contains information about a route has an instance that is simply the IP address of the route's destination. Other variables have more complex rules for forming instances. The variable name uniquely identifies a group of related data, while the variable instance is a unique name for a particular datum within the group. Thus, `1.3.6.1.2.1.4.21.1.10` is the name of the variable whose instances are route ages, while `1.3.6.1.2.1.4.21.1.10.127.50.50.50` is the name of the instance that contains the age of the route to a host with IP address `127.50.50.50`.

Understanding Terminology Related to Management Information Base (MIB) Variables

RFCs 1065 and 1066 define the Management Information Base (MIB) as an object-oriented database. They refer to the node names as Object Identifiers. Most nodes also have descriptive textual names called Object Descriptors. The Object Descriptors are convenient aliases, but SNMP request packets refer to variable instances only by Object Identifier. Variable names and variable instances are both denoted by Object Identifiers or Object Descriptors. To clearly distinguish the four possible combinations, the following non-RFC terminology is used here, in particular in describing the syntax and use of the API subroutines:

Terminology

Non-RFC Terminology	RFC Terminology	Example
Text-format variable name (denotes the descriptive textual name of a variable)	Object Descriptor of a variable	`sysDescr`
Numeric-format variable name (denotes a variable name expressed as a sequence of decimal numbers separated by periods)	Object Identifier of a variable	`1.3.6.1.2.1.1.1`
Text-format instance ID (denotes a text-format variable name qualified by an instance)	Object Descriptor of a variable with an instance appended	`sysDescr.0`
Numeric-format instance ID (denotes a numeric-format variable name qualified by an instance)	Object Identifier of a variable with an instance appended	`1.3.6.1.2.1.1.1.0`

Instance IDs are just variable names with an instance appended. A variable name refers to a set of related data, while an instance ID refers to a specific datum from the set.

For information on the API subroutines, see Using the SNMP API Subroutine Library on page 6–10 and Alphabetic List of API Subroutines on page 6–12.

Using the Management Information Base (MIB) Database

Network management can be passive or active. Passive management involves the collection of statistical data so that the network activity of each host can be profiled. Every variable in the Internet-standard MIB has a value that can be queried and used for this purpose. Active network management involves the use of a subset of MIB variables that are designated read-write. When an agent is instructed to modify the value of one of these variables, an action is taken on the agent's host as a side effect. For example, a request to set ifAdminStatus.3 to the value 2 has the side effect of disabling the network adapter card whose ifIndex is 3.

Requests to read or change variable values are generated by monitor applications. There are three kinds of requests:

get Returns the value of the specified variable instance.

get-next Returns the value of the variable instance following the specified instance.

set Modifies the value of the specified variable instance.

Requests are encoded according to the ISO ASN.1 CCITT standard for data representation (ISO document DIS 8825). Each get request contains a list of pairs of variable instances and variable values. This is called the variable binding list. The variable values are empty when the request is transmitted. The values are filled in by the receiving agent and the entire binding list is copied into a response packet for transmission back to the monitor. If the request is a set request, the request packet also contains a list of variable values. These values are copied into the binding list when the response is generated. If an error occurs, the agent immediately stops processing the request packet, copies the partially processed binding list into the response packet, and transmits it with an error code and the index of the binding that caused the error.

The get-next request deserves special consideration. It is designed to navigate the entire Internet-standard MIB subtree. Since all instance IDs are sequences of numbers, they can be ordered. The first four instance IDs are:

1.3.6.1.2.1.1.1.0	sysDescr.0
1.3.6.1.2.1.1.2.0	sysObjectId.0
1.3.6.1.2.1.1.3.0	sysUpTime.0
1.3.6.1.2.1.2.1.0	ifNumber.0

A get-next request for sysUpTime.0 returns a binding list containing the following pair: (ifNumber.0, *Value*). Instance IDs are somewhat like decimal number representations, with the digits to the right increasing more rapidly than the digits on the left. Unlike decimal numbers, the digits have no real base. The possible values for each digit are determined by the RFCs and the instances that are appended to the variable names. The important feature of the get-next request is that it allows a traversal of the whole tree, even though instances are not known.

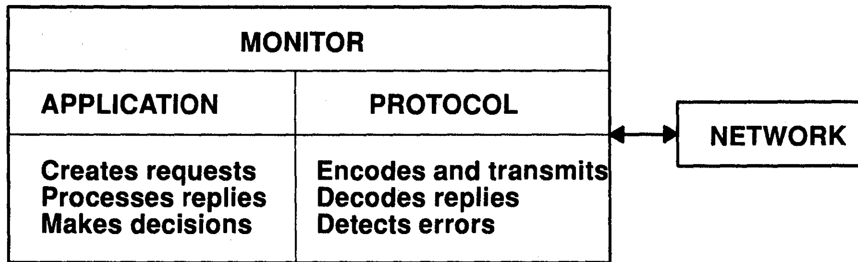
The following example is an illustration of the algorithm, not of actual code:

```
struct binding {
  char instance[length1];
  char value[length2];
  }bindlist[maxlistsize];
bindlist[0] = get(sysDescr.0);
for (i = 1; i < maxlistsize && bindlist[i-1].instance != NULL; i++)
{
  bindlist[i] = get_next(bindlist[i-1].instance);
  }
```

The fictitious get and get-next functions in this example return a single binding pair, which is stored in an array of bindings. Each get-next request uses the instance returned by the previous request. By daisy-chaining in this way, the entire MIB database is traversed.

Understanding How a Monitor Functions

Monitors are the clients in the client/server relationship. They are divided into two functional layers as shown in the following figure:

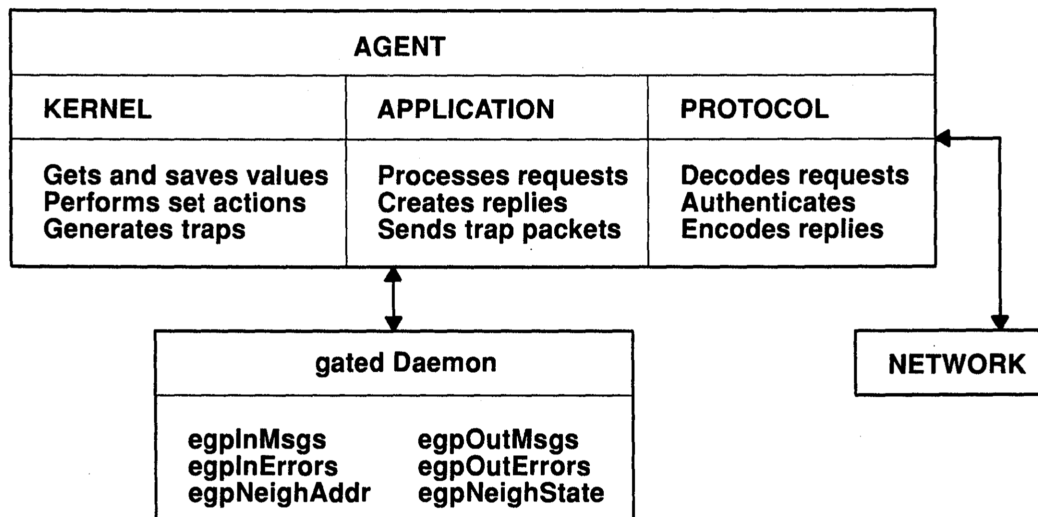


The protocol layer accepts requests from the application layer, encodes them in ASN.1 format, and transmits them on the network. It receives and decodes replies and trap packets, detects erroneous packets, and passes the data up to the application layer.

The application layer does the real work of the monitor. It decides when to generate requests for variable values and what to do with the results. A monitor may perform a merely passive statistics-gathering function, or it may attempt to actively manage the network by setting new values in read-write variables on some hosts. For example, a network interface may be enabled or disabled by means of the ifAdminStatus variable. The variables in the ipRoute family can be used to download kernel route tables, using data obtained from a router.

Understanding How an Agent Functions

Agents are the servers in the client/server relationship. Agents listen on well-known port 161 for request packets from monitors. In addition to the protocol and application layers, agents must also communicate with the operating system kernel. Most of the information in the Internet-standard MIB is maintained by kernel processes. The actions associated with a set request are often implemented as ioctl commands. In addition, the kernel may generate asynchronous notifications called traps. Some MIB information may be managed by another application, such as the **gated** daemon. The following figure illustrates the function of an agent:



One of the tasks of the protocol layer is to authenticate requests. This is optional and not all agents implement this task. If the protocol layer authenticates requests, the community name included in every request packet is used to determine what access privileges the sender has. The community name might be used to reject all requests (if it is an unknown name), restrict the sender's view of the database, or reject set requests from some senders. A monitor might belong to many different communities, within each of which it has a different set of access privileges granted by the agents. A monitor might generate or forward requests for other processes, using different community names for each.

Traps

The kernel may generate asynchronous event notifications called traps. For example, if an interface adapter fails, the kernel may detect this and generate a link-down trap (in some implementations, the agent may detect the condition). Other applications may generate traps. For example, the **gated** daemon generates an egp neighbor-loss trap whenever it puts an EGP (Exterior Gateway Protocol) neighbor into the down state. The agent itself generates traps (cold-start, warm-start) when it initializes, and when authentication fails (authentication-failure). Each agent has a list of hosts to which traps should be sent. The hosts are assumed to be listening on well-known port 162 for trap packets.

Working with Management Information Base (MIB) Variables

The **xgmon** client (manager) obtains its description of the Management Information Base (MIB) variables (described by RFC 1066) from the **mib_desc** file found in the /**etc** directory. This file has the following format:

Format of the mib_desc File			
Name	**Object ID**	**Type**	**TTL**
sysDescr	1.3.6.1.2.1.1.1.	string	900

The fields are separated by spaces or tabs, and contain the following information:

Name field Holds the text description of the object.

Object ID field Is the object identifier prefix assigned to the object class. The prefix is entered with a trailing . (dot) at the point just prior to where an instance would be specified.

Type field Denotes the type of the object as one of the following:

- Number
- String
- Object
- Internet
- Counter
- Gauge
- Ticks.

TTL field Contains the time-to-live value. This is the amount of time, in seconds, that the **xgmon** program should consider the data as valid. As long as a particular value is considered valid, the **xgmon** program will not generate a query to obtain it. In some situations, network traffic can be reduced by increasing the time-to-live value for a particular variable class. The time-to-live value should never be less than one second.

The Network Management program diskettes include a **mib_desc** file that contains all the variables as specified in RFC 1066.

The following intrinsic functions have parameters that specify MIB object IDs: the **base_type** intrinsic function, **gw_var** intrinsic function, **real_type** intrinsic function, and **snmp_var** intrinsic function.

The following library commands have parameters that specify MIB object IDs: the **snmp_get** library command, **snmp_next** library command, and **snmp_set** library command.

Using the SNMP API Subroutine Library

A C programmer can create SNMP manager applications by writing C programs that communicate with an SNMP agent. A set of library subroutines are provided for this interface. These library subroutines are stored in the **/usr/lib/libsnmp.a** file. A library required by the **libsnmp.a** file is stored in the **/usr/lib/libisode.a** file. This software is derived in part from the ISO Development Environment (ISODE). IBM acknowledges source author Marshall Rose and the following institutions for their role in its development: The Northrup Corporation and The Wollongong Group.

The **/usr/lib/libsnmp.a** and **/usr/lib/libisode.a** files must be linked with the user's application object files at link time.

Some of the API library subroutines access an additional file, the **/etc/mib_desc** file, which defines Management Information Base (MIB) variables as described in RFC 1066. The Simple Network Management Protocol (SNMP) is explained in RFC 1098.

To install the SNMP API library subroutines, place the IBM RISC System/6000 Network Management LPP diskette #1 in the diskette drive. Type:

```
installp -F netmgr.api
```

and press Enter.

External Variables

All applications that use the API library subroutines should include the following definitions:

```
extern int SNMP_port;   /* Socket returned by create_SNMP_port()  */
extern sockaddr_in snmp_dest;
                        /* Socket address from create_SNMP_port() */
extern char *SNMP_errormsg[];
                        /* Contains messages for SNMP Return codes */
```

In addition, the following external variables may be declared:

```
#include <sys/time.h>
extern struct timeval SNMP_timeout;
                       /* Retry interval - default 5 seconds  */
extern int max_SNMP_retries;
                       /* Maximum retry number - default is 3 */
extern int io_debug;   /* io_debug == 1 means "trace packets" */
```

The retry interval can be reset to 10 seconds, and the number of retries to 5, with the following instructions:

```
max_SNMP_retries = 5;
SNMP_timeout.tv_sec = 10;
```

Writing Applications Using the SNMP API Library

All applications that issue get, get-next, or set requests should have the following general format:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>

extern int SNMP_port;
extern sockaddr_in snmp_dest;
extern char *SNMP_errormsg[];

main(argc,argv)
int argc;
char *argv[];
{
unsigned long addr;     /* Binary IP address of agent host */
int      num_reqs;      /* Number of instance IDs in the request
                           packet */
char     rpacket[1024]; /* Buffer for ASN—encoded SNMP request
                           packet */
char     *req[50];      /* Array of up to 50 pointers to instance
                           IDs */
char     *sval[50];     /* Array of up to 50 pointers to set
                           values */
char     *community;    /* Community name required for
                           authorization */
int      len,rc;
...
rc = create_SNMP_port(addr); /* Bind a socket for use with SNMP */
if (rc < 0) exit(1);
...
len = make_SNMP_request(1,community,num_reqs,req,0,rpacket,
                        sizeof(rpacket));
if (len < 0) exit(2);        /* len > 0 is actual length of packet */
rc = send_recv_SNMP_packet(SNMP_port,&snmp_dest,rpacket,len);
if (rc < 0) exit(3);         /* No response received */
if (rc > 0) {                /* SNMP error code from agent */
        fprintf("%s\n",SNMP_errormsg[rc]);
        exit(4)
}
}
```

In this example, `make_SNMP_request` is asked to send a get request. If the first parameter is `2`, the request is a get-next. If the first parameter is `3`, it is a set request. In that case, the fifth parameter (`0` in the example above) would be `sva`, a pointer to an array of pointers to the set values corresponding to the instance IDs in the array `req`.

Replacing the API Library Routines

The **send_recv_SNMP_packet** subroutine calls the **parse_SNMP_packet** subroutine to analyze and process the get-response packet returned by the agent. An application that listens on port 162 for traps generated by agents might call the **parse_SNMP_packet** subroutine directly to process the trap data. After checking the packet for errors, the **parse_SNMP_packet** subroutine calls either the **save_SNMP_var** subroutine to process get-response packets, or the **save_SNMP_trap** subroutine to process trap data. The default action of the library versions of these subroutines is to print the formatted data to standard output.

These subroutines may be replaced by user-written subroutines that conform to the documented entry protocols. A replacement subroutine might save trap data in a database or forward it to another process. Remember that the **save_SNMP_var** subroutine is called once for each variable binding in a get-response packet, and the **save_SNMP_trap** subroutine is called once for each trap packet.

The **create_SNMP_port** subroutine does not support multiple concurrent data streams to agents. One way to reach many agents is to close the port designated by the *SNMP_port* variable after each use, and call the **create_SNMP_port** subroutine to reopen it for the next request. Another way to do it is to ignore the **create_SNMP_port** subroutine, and write application code to create a separate socket and corresponding socket address for each data stream. However, the **send_recv_SNMP_packet** subroutine uses the **sendto** subroutine to send the request, and then immediately calls the **select** subroutine to wait for a response. If true concurrence is needed, the application should also replace the **send_recv_SNMP_packet** subroutine.

You may wish to consult the following information: the **parse_SNMP_packet** subroutine, the **save_SNMP_trap** subroutine, the **save_SNMP_var** subroutine, the **send_recv_SNMP_packet** subroutine, and the **snmpd** command.

Alphabetic List of API Subroutines

create_SNMP_port — Creates a UDP socket to communicate with an SNMP agent.

extract_SNMP_name — Extracts the variable name portion of a numeric-format instance ID.

get_MIB_base_type — Returns a value indicating the base type of a Management Information Base (MIB) variable.

get_MIB_name — Returns the text name of a Management Information Base (MIB) variable.

get_MIB_variable_type — Returns a value indicating the variable type of a Management Information Base (MIB) variable.

lookup_addr — Returns the text name of a host.

lookup_host — Returns the Internet address of a host.

lookup_SNMP_group — Finds the set of all numeric-format variable names that contain a given text string as prefix.

lookup_SNMP_name — Returns the numeric-format name of a Management Information Base (MIB) variable.

make_SNMP_request — Encodes an SNMP request.

parse_SNMP_packet Decodes an SNMP packet.

save_SNMP_trap Stores SNMP trap data.

save_SNMP_var Stores retrieved SNMP variable data.

send_recv_SNMP_packet
Sends a query to and awaits a response from an SNMP agent.

SNMP_errormsg Stores SNMP error messages.

Understanding the SNMP Daemon

The SNMP daemon is a background server process that may be run on any AIX 3.0 TCP/IP host. The daemon, acting as SNMP agent, receives, authenticates, and processes SNMP requests from monitor applications. Read Understanding the Simple Network Management Protocol (SNMP) on page 6–3, Understanding How a Monitor Functions on page 6–8, and Understanding How an Agent Functions on page 6–8 for more detailed information on agent and monitor functions.

Note: The terms "SNMP daemon," "SNMP agent," and "agent" are used interchangeably.

TCP/IP support must be available before the SNMP daemon can be run, so the **inetd** process must be started before the **snmpd** command is issued. In addition, the lo0 device must be configured with the following command:

```
/etc/ifconfig lo0 loopback up
```

This software is derived in part from the ISO Development Environment (ISODE). IBM acknowledges source author Marshall Rose and the following institutions for their role in its development: The Northrup Corporation and The Wollongong Group.

Configuring the SNMP Daemon

The SNMP daemon will attempt to bind sockets to three ports, which must be defined in the **/etc/services** file as follows:

```
snmp          161/udp
mib           port#/udp
snmp–conf     port#/udp
```

The `snmp` service must be assigned port `161`, as required by RFC 1098. The two remaining services may be assigned any unused port number, and are used for internal purposes. It is recommended that the `mib` and `snmp–conf` services be assigned values greater than `1023`. The assigned ports must be available before the SNMP daemon can run.

The **/etc/services** file shipped with AIX 3.0 assigns default values to each of these services.

Two configuration files are used by the SNMP daemon:

- The **/etc/snmptrap.dest** file, which contains a list of hosts that are to receive trap packets.

  Note: The **/etc/snmptrap.dest** file contains unencrypted community names, so read permission should be granted only to members of the system group.

- The **/etc/snmpd.pw** file, which contains information used to authenticate requests and determine request attributes. For more information on this file, see the **smpl.pwinput** file and the **mksnmppw** command.

Understanding SNMP Daemon Processing

Message Processing and Authentication

All requests, traps, and responses are transmitted in the form of ASN.1-encoded messages. A message, as defined by RFC 1098, has the following structure:

Version Community PDU

where *Version* is the SNMP version (currently version 1), *Community* is the community name, and *PDU* is the Protocol Data Unit that contains the SNMP request, response, or trap data. A PDU is also encoded according to ASN.1 rules.

The SNMP daemon receives and transmits all SNMP protocol messages via the TCP/IP UDP datagram protocol. Requests are accepted on well-known port 161, and traps are transmitted to hosts identified in the **/etc/snmptrap.dest** file at the well-known port 162.

When a request is received, the source IP address and the community name are checked against the **/etc/snmpd.pw** file. If no matching entry is found, the request is ignored. If a matching entry is found, access is allowed according to the attributes assigned from the **/etc/snmpd.pw** file. Both the message and the PDU must be encoded according to ASN.1 rules.

The authentication scheme described above is not intended to provide full security. If the SNMP daemon is used only for get and get-next requests, security may not be a problem. If set requests are allowed, the set privilege may be restricted.

See the **snmpl.pwinput** file and the **mksnmppw** command for further information.

Request Processing

There are three types of request PDUs that may be received by the SNMP daemon. The request types are defined in RFC 1098, and the PDUs all have the following format:

Request PDU Format			
request-id	**error-status**	**error-index**	**variable-bindings**
GET	0	0	*VarBindList*
GET-NEXT	0	0	*VarBindList*
SET	0	0	*VarBindList*

The **request-id** field identifies the nature of the request; the error-status field and error-index field are unused and must be set to 0 (zero); and the variable-bindings field contains a variable-length list of numeric-format instance IDs whose values are being requested. If the value of the request-id field is SET, the variable-bindings field is a list of pairs of instance IDs and values.

Read Using the Management Information Base (MIB) Database on page 6–6 for a discussion of the three request types.

Response Processing

Response PDUs have nearly the same format as request PDUs:

Response PDU Format			
request-id	**error-status**	**error-index**	**variable-bindings**
GET-RESPONSE	*ErrorStatus*	*ErrorIndex*	*VarBindList*

If the request was successfully processed, the values for the error-status and error-index field are 0 (zero), and the variable-bindings field contains a complete list of pairs of instance IDs and values.

If any instance ID in the variable-bindings field of the request PDU was not successfully processed, the SNMP agent stops processing, writes the index of the failing instance ID into the error-index field, records an error code in the error-status field, and copies the partially completed result list into the variable-bindings field.

RFC 1098 defines the following values for the error-status field:

Values for the Error-Status Field		
Value	**Value**	**Explanation**
noError	0	Processing successfully completed (error-index is zero).
tooBig	1	The size of the response PDU would exceed an implementation-defined limit (error-index is zero).
noSuchName	2	An instance ID does not exist for GET and SET request types, or has no successor in the MIB for GET-NEXT requests (non-zero error-index).
badValue	3	For SET requests only, a specified value is syntactically incompatible with the type attribute of the corresponding instance ID (non-zero error-index).
readOnly	4	For SET requests only, an attempt to modify a read-only MIB variable is rejected (non-zero error-index).
genErr	5	An implementation-defined error occurred (non-zero error-index); for example, an attempt to assign a value that exceeds implementation limits.

Trap Processing

Trap PDUs are defined by RFC 1098 to have the following format:

Trap PDU Format					
enterprise	**agent-address**	**generic-trap**	**specific-trap**	**time-stamp**	**variable-bindings**
Object ID	*Integer*	*Integer*	*Integer*	*TimeTicks*	*VarBindList*

where the fields are used as follows:

enterprise
The Object Identifier assigned to the vendor implementing the agent. This is the value of the *sysObjectID* variable, and it is unique for each implementor of an SNMP agent. The value assigned to this implementation of the agent is 3.6.1.4.1.2.2.1.2.7.

agent-address
IP address of the object generating the trap.

generic-trap
Integer as follows:

0	coldStart
1	warmStart
2	linkDown
3	linkUp
4	authenticationFailure
5	egpNeighborLoss
6	enterpriseSpecific

specific-trap
Unused, reserved for future development.

time-stamp
Elapsed time, in hundredths of a second, from the last reinitialization of the agent to the event generating the trap.

variable-bindings
Extra information, dependent on generic-trap type.

The generic-trap values indicate that certain system events have been detected:

coldStart
The agent is reinitializing. Configuration data or MIB variable values, or both, may have changed. Measurement epochs should be restarted.

warmStart
Not implemented; reserved for future use.

linkDown
The agent has detected that a communications interface has been disabled.

linkUp
The agent has detected that a communications interface has been enabled.

authenticationFailure
A message was received that could not be authenticated.

egpNeighborLoss
An EGP neighbor was lost. This value is only generated when the agent is running on a host that runs the **gated** daemon using the Exterior Gateway Protocol (EGP).

enterpriseSpecific
Not implemented; reserved for future use.

The linkDown and linkUp traps contain a single instance ID/value pair in the variable-bindings list. The instance ID identifies the ifIndex of the adapter that was disabled or enabled, and the value is the ifIndex value. The trap for egpNeighborLoss also contains a binding consisting of the instance ID and value of egpNeighAddr for the lost neighbor.

When an interface is disabled, MIB variable values for that interface may be lost. When an interface is enabled, a linkUp trap is generated. A flag in the **/etc/snmptrap.dest** file may be set for each destination host. This causes a coldStart trap to be generated as well. The purpose is to guarantee that the receiving host understands that crucial MIB variable values may have changed. In particular, measurement epochs should be restarted.

Understanding SNMP Daemon Support for the EGP Family of MIB Variables

If the agent host is running the **gated** daemon with Exterior Gateway Protocol (EGP) support enabled, the following MIB variables are available:

egpInMsgs
The number of EGP protocol messages received by the **gated** daemon running on the agent's host.

egpInErrors The number of EGP messages received in error.

egpOutMsgs The number of EGP messages transmitted by the **gated** daemon running on the agent's host.

egpOutErrors The number of EGP messages that could not be sent by the agent host's **gated** daemon due to resource limitations.

egpNeighState The state of an EGP peer acquired by the agent host's **gated** daemon:

1	idle
2	acquisition
3	down
4	up
5	cease

egpNeighAddr The IP address of an EGP peer acquired by the agent host's **gated** daemon.

If the **gated** daemon is not running, get or get-next requests for the values of these variables will return the noSuchName error response code.

To communicate properly with the SNMP agent, the **/etc/services** file must have an assigned UDP port for the mib service. In addition, the **/etc/gated.conf** file should contain the following statement:

```
SNMP yes
```

Warning: The **gated** daemon manages the kernel routing tables. The ipRouteNextHop and ipRouteType MIB variables may also be used to manage kernel routing tables. The SNMP agent does not coordinate with the **gated** daemon in any way. Therefore, it is *strongly recommended* that neither the **gated** nor **routed** daemons are started if the SNMP agent is to manage routing tables.

For information on the ipRouteNextHop and ipRouteType MIB variables, see the following section.

Understanding SNMP Daemon Support for SET Request Processing

The **snmpd** daemon supports set requests for five read/write MIB variables:

atPhysAddress The hardware address portion of an address table binding on the agent's host (an entry in the arp table).

ifAdminStatus The state of an interface adapter on the agent's host (up or down).

ipDefaultTtl The default time-to-live value inserted into IP headers of datagrams originated by the agent's host.

ipRouteNextHop The gateway by which a destination IP address can be reached from the agent's host (an entry in the route table).

ipRouteType The state of a route table entry on the agent's host (used to delete entries).

The variables require instances and values as shown in the following table:

Instances and Values of Variables			
Name	**Instance**	**Value**	**Action**
atPhysAddress	f.1.n.n.n.n	hh:hh:hh:hh:hh:hh	For the interface with ifIndex f, any existing arp table binding for IP address n.n.n.n is replaced with the binding (n.n.n.n, hh:hh:hh:hh:hh:hh). If a binding did not exist, the new binding is added. hh:hh:hh:hh:hh:hh is a twelve hexadecimal digit hardware address.
		00:00:00:00:00:00	Any arp binding on ifIndex f for IP address n.n.n.n is deleted.
ifAdminStatus	f	1	The interface adapter with ifIndex f is enabled.
		2	The interface adapter with ifIndex f is disabled.
ipDefaultTtl	0	n	The default time-to-live value used by IP protocol support is set to the integer n.
ipRoutNextHop	n.n.n.n	m.m.m.m	A route table entry to reach network n.n.n.n via gateway m.m.m.m is added to the route table. The host portion of the IP address n.n.n.n must be zero to indicate a network address.
ipRouteType	h.h.h.h	2	Any route to host IP address h.h.h.h is deleted.
	n.n.n.n	2	Any route to host IP address n.n.n.n is deleted.

Note: It is *strongly recommended* that the atPhysAddress variable only be used to create address table entries for hosts that do not participate in the Address Resolution Protocol (ARP). Most hosts on broadcast networks (Ethernet, Token-Ring, 802.3) have ARP capability. Reliance on ARP will result in more efficient and responsive address table management.

Examples

The following examples use the **snmp_set** library command for the **xgmon** command.

1.

```
snmp_set host1 atPhysAddress.2.1.255.255.255.255=02:60:8c:ab:cd:ef
```

This command creates a new ARP table binding on `host1`, associated with a network interface adapter with ifIndex 2. The binding is (`255.255.255.255`, `02:60:8c:ab:cd:ef`). This means that the IP address `255.255.255.255` is associated with the hardware address `02:60:8c:ab:cd:ef` on the physical network reached by the interface adapter with ifIndex 2. If an arp binding for `255.255.255.255` already exists, it is replaced. (See note above.)

2.

```
snmp_set host1 ifAdminStatus.2=2
```

This command disables the network interface adapter that has ifIndex 2. If the assigned value is 1, the interface adapter is enabled.

3.

```
snmp_set host1 ipDefaultTtl=50
```

This command allows an IP datagram using default time-to-live to pass through up to `50` gateways before being discarded.

4.

```
snmp_set host1 ipRouteNextHop.255.255.255.255=128.0.0.1
```

This command creates a route to host `255.255.255.255` via the gateway host `128.0.0.1`. The ipRouteType value is 4 (remote).

5.

```
snmp_set host1 ipRouteNextHop.255.255.255.0=128.0.0.1
```

This command creates a route to the class C network `255.255.255` via the gateway host `128.0.0.1`. Note that the host part of the address must be 0 (zero) to indicate a network address.

6.

```
snmp_set host1 ipRouteType.255.255.255.255=2
```

This command deletes any route to host `255.255.255.255`.

Understanding SNMP Daemon RFC Conformance

RFC 1098 requires that each set request variable assignment "should be effected as if simultaneously set with respect to all other assignments specified in the same message" (page 25). This means that a set request with multiple instance ID/value pairs should be processed in an all-or-none fashion: either all the new values are assigned without error, or else none of the variables in the request have modified values. This requirement is also known as *atomic commit with rollback*. The RFC does not address the problems of order dependency or consistency.

Atomic commit is not currently implemented. This should not present a problem in the use of the five read-write MIB variables currently allowed. The possible actions that can be taken by modifying these variables are:

- Adding a route
- Deleting a route
- Enabling an adapter
- Disabling an adapter
- Adding an arp table entry
- Deleting an arp table entry
- Changing default time-to-live.

Each of these actions is accomplished by modifying a single variable. There are no dependencies between the variables. Therefore, a set request containing multiple bindings is equivalent to a sequence of single-binding set requests, and will be processed as such.

RFC 1066 describes all ten variables in the ipRouteEntry table as read-write. As described above, set support is only implemented for ipRouteType and ipRouteNextHop (see note 2). In order to accept set requests that may specify several unsupported route attributes (such as the ipRouteMetric1 or the ipRouteProto attributes), set requests for these unsupported variables are accepted. No error response is returned to the request originator, but a subsequent get request will show that the original values have been retained.

Note that ordering dependencies may exist, such as in:

```
snmp_set host1 atPhysAddress.f.1.n.n.n.n=hh:hh:hh:hh:hh:hh
ifAdminStatus.f=1
```

```
snmp_set host1 iproutenexthop.n.n.n.n=m.m.m.m ifAdminStatus.f=1
```

where the adapter with ifIndex f must be enabled before an address table entry can be associated with it, or a route can be established that reaches a gateway via that adapter. In these cases, the order of the variables is significant, and must be the reverse of that shown above. If the RFC 1098 atomic commit policy were followed exactly, the set requests described above would have undefined effects.

Consistency problems are not addressed by RFC 1098. For example, the request

```
snmp_set host1 ipRouteNextHop.n.n.n.n=m.m.m.m ipRouteType.n.n.n.n=2
```

causes a route to be added and then immediately deleted. If the order of the variables is reversed, any existing route is deleted, and then the new route is added. If the RFC 1098 atomic commit policy were followed exactly, the set request described above would have an undefined effect.

Understanding SNMP Daemon Implementation Restrictions

The current implementation of the agent does *not* have:

- National Language Support (NLS)
- System Resource Controller (SRC) support
- Proxy agent support
- Support for restricted MIB database views
- Non-trivial authentication support
- Coordination of routing table management with the **gated** or **routed** daemons. It is *strongly recommended* that **gated** and **routed** daemons *not* be run if the SNMP agent is used to manage kernel route tables.

There is no support for set requests to modify values of read-write MIB variables except those described in Understanding SNMP Daemon Support for SET Request Processing.

Not all values defined in RFC 1066 are supported, as illustrated in the following table:

Support for Values of Read-Write MIB Variables Defined in RFC 1066

Name	Value	Support
ifAdminStatus	1 (up)	Supported for get and set requests
	2 (down)	Supported for get and set requests
	3 (testing)	Not supported for get or set requests
ipRouteType	1 (other)	Not supported for get or set requests
	2 (invalid)	Supported for set requests only (see note 1)
	3 (direct)	Supported for get requests only (see note 2)
	4 (remote)	Supported for get requests only (see note 3)

Notes:

1. A set request for ipRouteType 2 (invalid) causes deletion of a route. A get request will obviously never return this value for an existing route.
2. A value for ipRouteType is always assigned when a route entry is added to the kernel route table. When the **ifconfig** command is issued on the host to enable an adapter, a route to the adapter of ipRouteType 3 (direct) is automatically created. This is the only way to create a direct route.
3. A value for ipRouteType of 4 (remote) indicates a route to a destination host via a gateway host. The gateway host is not the local host, but is directly reachable on a physically connected network. All routes added via the ipRouteNextHop variable are of this type.

The following implementation restrictions should be observed:

- An interface must be enabled by a locally issued **ifconfig** command before it can be enabled or disabled by the ifAdminStatus variable. There is no way to associate an IP address with an interface except with a locally issued **ifconfig** command. The ifIndex value is assigned when the **ifconfig** command is issued.
- An interface may only be detached by a locally issued **ifconfig** command. Another **ifconfig** command is required to enable the adapter before it can be managed via the ifAdminStatus variable.

- The SNMP agent does not provide a way to alter the IP address associated with an interface adapter. This must be done with a locally issued **ifconfig** command.
- The SNMP agent will not search a route table hash chain to a depth of greater than 100 during set processing. If a hash chain longer than 100 is detected, set requests for the ipRouteType and ipRouteNextHop variables will be rejected with a genErr response. Other conditions may cause this response.

  If a set request for ipRouteNextHop receives a genErr response code, the condition that caused it may be transitory. The monitor application should reissue the set request at least once.
- The maximum message size that can be sent or received is 2048 bytes.

Understanding the SNMP Command Line Manager

The SNMP Command Line Manager consists of three executable commands:

- **/usr/bin/snmp_get**
- **/usr/bin/snmp_next**
- **/usr/bin/snmp_set**

These commands are executed on the AIX command line. They obtain or modify information about a host by sending GET, GET-NEXT, or SET requests to the SNMP agent running on that host.

The **/etc/mib_desc** file describes the supported MIB variables that are defined in RFC 1066. The requests are defined in RFC 1098.

To install the SNMP Command Line Manager, place the IBM RISC System/6000 Network Management LPP diskette #1 in the diskette drive. Type:

```
installp -F netmgr.clm
```

and press Enter.

This software is derived in part from the ISO Development Environment (ISODE). IBM acknowledges source author Marshall Rose and the following institutions for their role in its development: The Northrup Corporation and The Wollongong Group.

Understanding the xgmon Programming Utility

The **xgmon** programming utility is used to write extensions to the **xgmon** client (manager). It is both a subset and a superset of the C programming language. As a subset it uses syntax similar to C, though it does not support the complete language. As a superset, it supports operations (for example, comparison and addition) on types such as strings.

The **xgmon** program contains a compiler that translates programs written in the **xgmon** programming utility into object code that is interpreted by virtual G machines (VGMs). The **xgmon** application oversees the operation of the VGMs. Each VGM works on its own program. More than one program can be executing at the same time because **xgmon** is able to control more than one virtual G machine.

The **xgmon** program is essentially a passive entity. Any required monitoring algorithms are embedded within library commands, which are programs written in the **xgmon** programming utility. These programs are stored as source code.

There are two kinds of commands available to the user at the **xgmon** command prompt (>):

System commands
: System commands are built into the client (manager), are generally used to control virtual G machines, and *cannot* be extended.

Library commands
: Library commands are programs written in the **xgmon** programming utility and *can* be extended. When a library command is referenced, the corresponding source code is compiled. The resulting object code is loaded into a virtual G machine, and the virtual G machine begins executing the command.

Since much of the **xgmon** programming utility looks like C language, C programmers can learn it easily. The **xgmon** programming utility is intended to support writing applications that are used to monitor TCP/IP-based networks.

The capabilities of the **xgmon** system can be extended by the programmer merely by writing a new program in the **xgmon** programming utility and adding this program to the directory of library commands.

Extending xgmon Intrinsic Functions

The **xgmon** program consists of intrinsic functions that are activated by library commands. The **xgmon** program has two distinct object files:

xgmon.o file — Object code of the basic **xgmon** program.

user_func.o file — Object code of the intrinsic functions defined by users.

A C programmer can extend the **xgmon** program by adding new intrinsic functions. These new intrinsic functions must be coded in C and must be included in the **user_func.c** file, which has been shipped in the **/usr/lpp/xgmon/bin** directory. The C programmer can create additional files to contain the source code of new intrinsic functions.

The C programmer must compile each file that contains the new intrinsic functions in order to create the corresponding object files. These object files can then be linked with **xgmon.o** to build the new extended **xgmon** program. For a full description of this procedure, see How to Create **xgmon** Intrinsic Functions on page 6–43.

A user-defined intrinsic function is called similar to a C program. It has two parameters. The first is the count of the number of arguments the function is passed. The second is an array, each element of which is the value of the corresponding argument. The value the C function returns will be the result of the intrinsic function as seen by the **xgmon** library program that is executing.

To override an existing intrinsic function, give the new function the same name as the function it replaces.

For lists of intrinsic functions, see Alphabetic List of Intrinsic Functions on page 6–37 and Functional List of Intrinsic Functions on page 6–39.

Creating xgmon Library Commands

To develop and test a library command, store the command source in a working directory that belongs to the user. After the new command is debugged, move the source code to the **/usr/lpp/xgmon/lib** directory so that it is included in the global list of library commands. This makes the new command available to all **xgmon** users.

You can add new library commands at any time and use them without having to terminate the current **xgmon** program. Similarly, you can correct bugs in library commands without stopping everything. This means that algorithms can be corrected without losing global state information that has already been collected. Thus, the failure of one library command does not necessarily incapacitate the rest of the monitoring system. This feature makes the monitoring system as available as the network it is monitoring. See How to Create **xgmon** Library Commands on page 6–46.

Programming Virtual G Machines (VGMs)

A virtual G machine is a section of memory that belongs to the **xgmon** program. The VGM is loaded with the object code generated by compiling an application (library program) written using the **xgmon** programming utility, and the data area for use by the application. The **xgmon** program interprets the object code in each VGM in a round-robin fashion, subdividing its own time slice.

A VGM executes **xgmon** library program object code produced by the **xgmon** internal compiler. For instance, when a user issues a library command, the virtual G machine executes the command and sends the output to the VGM standard output device. If **xgmon** is running under X11, this output device is a graphics window that is associated with that VGM. If **xgmon** is running in the ASCII mode, this output device is the standard output. However, it is possible to redirect the output from a VGM into a file.

If **xgmon** is running under X11, the VGM output window supports both text and graphics formatted output. A VGM can determine if the output is being sent to a window or to the standard output. Thus, a VGM is able to determine if the graphics function is available.

If **xgmon** is running in the ASCII mode, graphics are not supported.

A maximum of 6 VGMs can be activated when **xgmon** is invoked. Thus, if **xgmon** is running under X11, 6 VGM output windows can be displayed on the screen simultaneously. Having several VGMs available at the same time allows multiple, independent library commands to be in progress, thereby making it possible to receive output from several VGM processes simultaneously. Each VGM works on its own library program and is essentially independent of all other VGMs, although it is possible to implement some inter-machine communication if needed.

The **xgmon** program implements an abstraction of VGM architecture. The actual details of this abstraction are not critical as all programmer access to the VGM is via **xgmon** library programs. Several underlying characteristics of the VGM are, however, important to understand, such as:

- Formatting the VGM Windows, see page 6–25
- The VGM Run-time Environment, see page 6–30
- Working with VGM Data Types, see page 6–29
- Working with VGM Variables, see page 6–26.

See also the intrinsic functions for VGM control: the **aix_exec** intrinsic function, **alloc** intrinsic function, **ctime** intrinsic function, **exec** intrinsic function, **reuse_mem** intrinsic function, **time** intrinsic function, and **words_free** intrinsic function.

Understanding the Internal Database

Only a library program executed by a virtual G machine can make SNMP (Simple Network Management Protocol) requests. The **xgmon** program makes no requests of SNMP agents on its own. When a request is made, the server may or may not respond. Whenever the client (manager) receives a response from an SNMP agent, it stores the response in an internal database, making it available to all of the virtual G machines.

Variables are stored in the internal database temporarily. These variables have time stamps associated with them, and after a certain period of time these variables are discarded as invalid. As long as a variable in the database is believed to be valid, a request for that variable is satisfied from the internal database, and no query is sent to the remote host. This helps reduce the amount of traffic generated by the client (manager).

The MIB (Management Information Base) file lists each variable by text name and describes its object ID, type, and time-to-live value (in seconds). The name of this file is **/etc/mib_desc**. See Working with Management Information Base (MIB) Variables on page 6–9.

The following intrinsic functions have parameters that specify MIB object IDs: the **base_type** intrinsic function, **gw_var** intrinsic function, **real_type** intrinsic function, and **snmp_var** intrinsic function.

The following library commands have parameters that specify MIB object IDs: the **snmp_get** library command, **snmp_next** library command, and **snmp_set** library command.

Formatting the Virtual G Machine (VGM) Windows

Each virtual G machine (VGM) has a window you can write text to (using the **print** statement) and draw graphics to (using the **draw_line** and **draw_string** intrinsic functions).

To format a VGM window, use the following escape sequences and control codes.

Recognized Escape Sequences

The internal **xgmon** compiler recognizes the following escape sequences inside string constants:

Escape Sequences			
Escape Seq.	**Name**	**Dec. Value**	**Description**
\0	NUL	0	Null character, string delimiter
\t	HT	9	Horizontal Tab (tabs are set at every 8 characters)
\n	LF	10	Line Feed, New Line
\f	FF	12	Form Feed, New Page
\r	CR	13	Carriage Return
\'	"	34	Double Quote
\\	\	92	Back slash

Window Control Codes

The VGM output windows recognize the following control codes:

Window Control Codes	
Code	**Action**
7	Rings the bell.
HT	Advances the cursor to the next tab stop.
LF	Advances the cursor to the next line, column 1.
FF	Clears the window, moves the cursor to line 1, column 1.
CR	Returns the cursor to column 1.
27 l c	Positions the cursor to line l, column c.

Note: Column and row numbering start at 1.

Example

To start the cursor at column 1, row 1, enter:

```
print "%c%c%c", 27, 1, 1;  // cursor to upper left corner
```

Working with Virtual G Machine (VGM) Variables

Each virtual G machine (VGM) can have three kinds of variables:

Local variables Local to the specific VGM. In general, these are read/write variables.

Global variables Shared by all VGMs on a read-only basis.

Special local variables Values set by the **xgmon** program.

Local Variables

Local variables must be declared before use. The suggested convention is to declare them at the top of the library program. Global variables are defined by the system; user's library programs do not declare them. To declare local variables, include a list of variables in the library program prefixed with the desired data type, as demonstrated below:

```
int i,j;
string s,str2;
pointer ptr,ptr2;
```

Global Variables

In addition to the local variables defined for each VGM, a set of predefined global variables are maintained by the **xgmon** program. Global variables communicate information to VGMs. This information arrives from outside the client (manager), for instance, the arrival of trap information or the pressing of a mouse button.

Global Variables		
Name	**Type**	**Function**
selected_host	string	Name of the node or host on which the right mouse button was clicked.
selected_window	string	Name of the topology window the pointer was in when the button was pressed. The name of the root topology window is " "(null string).
selected_x	integer	x coordinate of pointer when button was pressed.
selected_y	integer	y coordinate of pointer when button was pressed.
element_mask	integer	Element mask of current display.
traps_pending	integer	Number of traps left in queue.
trap_host	string	Name of host from which the trap arrived.
trap_type	string	Type of trap received.
trap_userdata	string	Additional user-defined trap data.
ping_hosts	pointer	List of hosts that respond to the **ping** library command.
snmp_hosts	pointer	List of hosts that respond to an SNMP request.
snmp_passwords	pointer	Community names associated with the *snmp_hosts* variable.

Notes:

1. Global variables must be treated as read-only. If a virtual machine attempts a store operation outside of memory allocated to it, it will be stopped.
2. The *ping_hosts*, *snmp_hosts,* and *snmp_passwords* variables are lists whose ends are marked by the null string.
3. The *trap_type* variable indicates both the class of the trap, as well as its type. The class is indicated by the **snmp:** prefix (indicating an SNMP trap). The remainder of the string indicates the actual trap type. Thus, an SNMP link-up trap appears as **snmp:3**. The **xgmon** also generates "traps" when it changes the coloring of a display element. These traps are indicated by the **xgmon** prefix.

Trap Types

cold-start	snmp:0
warm-start	snmp:1
link-down	snmp:2
link-up	snmp:3
authentication-failure	snmp:4
EGP neighbor-loss	snmp:5
enterprise-specific	snmp:6

4. The *trap_userdata* variable contains additional information about the trap. It contains information that was provided in the original SNMP trap packet, such as the interface number in a link-up or down-trap, or the IP address (in dot notation) of the EGP peer in an EGP neighbor-loss trap. If the trap indicates an **xgmon** display element state change (for example, up, down, or unknown), this variable indicates the new state of the display element.
5. If the machine *hostid* of the agent generating the trap has not been set, the *trap_host* variable will default to `0.0.0.0`.

Special Local Variables

There are several special local variables that may be defined in an **xgmon** library program. These special local variables are set by the **xgmon** program. If these variables are defined, the underlying run-time environment updates them appropriately.

Special Local Variables		
Name	**Type**	**Function**
argc	integer	Number of arguments to the library program.
argv	pointer	Array of argument strings.
send_response	integer	Indicates response to a send request (0 = no error).
gw_var_size	integer	Size in bytes of the return value from the last call to the **gw_var** intrinsic function.
gw_var_name	string	Name of last variable received in response to a send request.

Notes:

1. The *send_response* variable is set to -1 if no response was received from the server or agent. Otherwise, this variable is set to the error code sent by the remote host (see the RFC 1098 for these values). A value of 0 indicates no error occurred. For information about the **send** statement, see Using Simple Statements on page 6–32.
2. The *gw_var_name* variable is used when making get-next requests. The name of the MIB variable the agent provided as a response to the query is made available in this variable. For information about the **send** statement and get-next requests, see Using Simple Statements on page 6–32.

Working with Virtual G Machine (VGM) Data Types

Virtual G machines (VGM) support three types of variable data: integers, strings, and pointers. Integers and strings are supported by the underlying VGM architecture. Depending on the context, an integer may be treated as an unsigned quantity. The basic unit of storage of a VGM is one word, which is 32 bits in length. A string is a null-terminated sequence of characters (identical to the C convention) with an index that starts at 1 (one).

The VGMs also support arrays, which lead to the pointer variable. Pointers are intended to refer to the base of a block of storage containing interesting data. Subscripting can be used to dereference the pointer and extract the data (subscripting is indicated with the use of variables). The data extracted may be treated as an integer, string, or pointer. Array references can only occur in the following situations:

- As a result of an assignment, as in:

  ```
  var[i] = expression;
  ```

- As the only component of an expression on the right hand side of an assignment statement, as in:

  ```
  var = var[i];
  ```

- As a parameter of an intrinsic function call, as in:

  ```
  pw = (string) password(argv[1]);
  ```

Note: No operations are permitted because the pointer has no type defined.

Strings are passed by reference, not value. The value stored in a string variable is a reference to the string data. That is, the string variable is a pointer, not the actual string data.

VGMs have a finite amount of memory available to them. As they work, they continue to use up this memory to store intermediate string expressions. The result is that, over time, a VGM uses up all of its available memory. Any long-running **xgmon** program has to recognize this and be prepared to detect an impending lack of free memory and to respond to it (see the **words_free** intrinsic function). VGMs do no garbage collection by default.

The **xgmon** program performs a form of hybrid garbage collection if you explicitly enable this function within an **xgmon** library program (see the **reuse_mem** intrinsic function) and the **reuse** system command has been issued to enable the hybrid garbage collector.

With garbage collection enabled, when a string assignment statement is executed, the storage space that the variable references is freed *prior* to the variable taking on its new value. The programmer must be careful when making string assignments because the underlying implementation passes references to strings, not the actual value. Thus, the following would cause problems if garbage collection is enabled:

```
var1 = "string expression";
var2 = var1;                /*  No problem yet.               */
var1 = "new expression";    /* Now var2 references a space that */
                           /*   as been freed.               */
```

However, a *copy* of the string can be created by concatenating the null string to the end of the string variable taking on the new value:

```
var1 = "string expression";
var2 = var1 + "";        /*   var2 now references a separate copy. */
var1 = "new expression"; /*   var2 is still valid.                 */
```

xgmon does not reclaim the storage used for intermediate string expressions. The programmer must explicitly store such expressions in string variables. Thus, instead of:

```
print "—naddress = %s—n", (string) dotaddr (addr);
```

Use:

```
print "—naddress = %s—n", (string) dotaddr (addr);
```

By following the guidelines outlined above, it is possible to write library programs that will achieve a steady state of storage utilization instead of growing without bound.

Understanding the Virtual G Machine (VGM) Run-Time Environment

The virtual G machines (VGMs) provide a simple, easily understood run-time environment. Each machine has an address space composed of the following segments:

Text segment Holds the object code

Data segment Holds local variables, string constants, dynamically allocated blocks, and intermediate string expressions generated during the evaluation of a string expression

Stack segment Holds parameters during evaluation of expressions.

Understanding the Structure of xgmon Library Programs

An **xgmon** library program is introduced by the phrase **start thread** and ends with the phrase **end thread.**

The following complete library program (equivalent to the **ping** library command) illustrates an **xgmon** library program:

```
// ping hostname
start thread
int argc;
pointer argv;

int ret_code;
string host;

    if (argc != 2) {
         print "usage:  %s hostname\n",argv[0];
         exit;
    }
    host = argv[1];
    ret_code = (int) ping(host);
    if (ret_code != -1) {
         print "%s responded, time=%d ms\n", host, ret_code;
    }
    if (ret_code == -1) {
         print "no response from %s\n",host;
    }
end thread
```

Comments

Comments in an **xgmon** program are introduced with:

(pound sign)

// (double slash)

— — (dash)

A comment extends to the end of the line. Comments can be anywhere in the **xgmon** library program source.

Supported Types

The **xgmon** programming utility supports three basic data types:

int	For integers
string	For strings
pointer	For use in implementing arrays.

The basic unit of storage in a virtual G machine (VGM) is one word (32 bits).

Declaring Variables

Variables must be declared before use. The suggested convention is to declare them at the top of the program following the **start thread** statement. Global variables are defined by the system; user programs do not declare them. To declare local variables and special local variables, include a list of variables in the prefix with the desired type, as demonstrated below:

```
int argc;
pointer argv;
int i,j;
string s,str2;
pointer ptr,ptr2;
```

xgmon Library Program Reserved Words

The following keywords are reserved by **xgmon** and cannot be used as variable names in **xgmon** library programs:

alternate	**are**	**asend**	**at**
define	**display**	**do**	**end**
exit	**for**	**from**	**get**
group	**if**	**in**	**int**
interfaces	**link**	**links**	**logical**
next	**no**	**node**	**nodes**
physica	**pointer**	**print**	**send**
set	**sleep**	**snmp**	**start**
string	**thread**	**to**	**under**
use	**while**	**window**	**with**

Using Simple Statements

There are several simple statements in the **xgmon** programming utility. As in the C language, the **;** (semicolon) is used as a statement terminator, not a statement separator. Several statements may be grouped as one by enclosing them in braces as follows:

```
{
   statement1;
   statement2;
}
```

Such a grouping is treated as one logical statement.

Assignment Statement

The most common statement is the assignment statement:

var_name = *expression*;

Sleep Statement

The **sleep** statement can be used to pause for a moment:

sleep *int_expression*; *// pause for specified seconds*

Exit Statement

The **exit** statement terminates the current program and unloads the virtual G machine (VGM) executing the program:

exit;

Send Statement

The **send** statement is used to send requests to a Simple Network Management Protocol (SNMP) agent. The *agent* parameter must be a string data type and may either be the Internet Protocol (IP) address in dot notation or the text name of the agent. The syntax of the **send** statement is as follows:

send snmp *req_type req_string* **to** *agent* (*community_string*);

send snmp set *req_string*, *value* **to** *agent* (*community_string*);

where the *req_type* parameter is either **get** or **get next** and the *community_string* parameter specifies the community name of the specified SNMP agent.

For SNMP get requests, the *req_string* parameter must be a numeric-format instance ID indicating the requested MIB variable (for example, 1.3.6.1.2.1.1.1.0). For SNMP get-next requests, the *req_string* parameter can be either a numeric-format variable name or a numeric-format instance ID.

Note: For an explanation of the terminology used, see Understanding Terminology Related to Management Information Base (MIB) Variables on page 6–6.

The *gw_var_name*, *gw_var_size,* and *send_response* special local variables are updated if declared.

Asend Statement

The **asend** (asynchronous send) statement is also used to send requests to an SNMP agent and functions in a similar fashion as the **send** statement. The **asend** statement differs from the **send** statement in that it does not wait for a response; therefore, no status information is made available. The *agent* parameter must be a string data type and may either be the Internet Protocol (IP) address in dot notation or the text name of the agent. The syntax of the **asend** statement is as follows:

asend snmp *req_type req_string* **to** *agent* (*community_string*);

asend snmp set *req_string*, *value* **to** *agent* (*community_string*);

where the *req_type* parameter is either **get** or **get next** and the *community_string* parameter specifies the community name of the specified SNMP agent.

For SNMP get requests, the *req_string* parameter must be a numeric-format instance ID indicating the requested MIB variable (for example, 1.3.6.1.2.1.1.1.0). For SNMP get-next requests, the *req_string* parameter can be either a numeric-format variable name or a numeric-format instance ID.

Note: For an explanation of the terminology used, see Understanding Terminology Related to Management Information Base (MIB) Variables on page 6–6.

The *gw_var_name* and *gw_var_size* special local variables are updated if declared, but the *send_response* special local variable is not.

Print Statement

The **print** statement is used to display formatted output on the status window associated with the virtual G machine. It can also be used to write output to a file that was previously opened with the **fopen** intrinsic function. The syntax of the **print** statement is as follows:

print *format_string* [,*arg1*]... [**to** *file_descriptor*];

Note: The format string can be specified as permitted by the **printf** subroutine.

Using Iteration and Conditional Statements

In addition to simple statements, the **xgmon** programming utility has looping and control statements.

If Statement

The **if** statement is like its C counterpart. If the conditional expression is nonzero, the statement is executed. The syntax of the **if** statement is as follows:

if (*int_expression*) *statement*

While Statement

Some looping may be performed using the **while** statement, which works like its C counterpart. If the conditional expression is nonzero, the statement is executed. The syntax of the **while** statement is as follows:

while (*int_expression*) *statement*

For Statement

It is possible to iterate over a list of expressions by using the **for** statement. The syntax of the **for** statement is as follows:

for *var_name* **in** {*expr1* [, *expr2*]... } **do** *statement*

The **for** statement starts with the last element and proceeds toward the first.

Using Expressions

The simplest expression is a constant. Integer constants and string constants are easy to understand, as illustrated below:

```
int i;
i = 2;
string s;
s = "hello";
```

You can build more complicated expressions with operators.

To obtain the desired evaluation order, use parentheses to group the expressions.

Expressions are not limited to operations on constants. Expressions can reference variables and call intrinsic functions.

Pointer variables may be dereferenced using subscripting. A pointer variable points to an array of elements. Each element in the array may be an integer, string, or another pointer. Subscripting is indicated with the use of brackets as follows:

var_name [*int_expression*]

Using Operators

The **xgmon** programming utility supports several operators:

Operators		
Type	**Operator**	**Result**
int	+	sum of two integers
int	–	difference of two integers
int	*	product of two integers
int	/	integer dividend of two integers
int	&&	bitwise AND of two integers
int	||	bitwise OR of two integers
string	+	concatenation of two strings

There are also several logical operators. A logical operator always produces an integer result. The value 0 represents false, and any nonzero value is treated as true.

Logical Operators		
Type	**Operator**	**Result**
int	||	logical OR of two operands
int	&&	logical AND of two operands
int	= =	true if operands are equal
int	! =	true if operands are not equal
int	<	true if op1 is less than op2
int	>	true if op1 is greater than op2
int	< =	true if op1 is less than or equal to op2
int	> =	true if op1 is greater than or equal to op2
string	= =	true if operands are equal
string	! =	true if operands are not equal
string	<	true if op1 is less than op2
string	>	true if op1 is greater than op2
string	< =	true if op1 is less or equal than op2
string	> =	true if op1 is greater than or equal to op2

Using Intrinsic Functions

There are several intrinsic functions built into the **xgmon** programming utility. Some functions are able to return values of different types. The most notable such function is the **gw_var** intrinsic function.

When an intrinsic function is referenced, the return value must be cast. Thus, all references to intrinsic functions are preceded by one of the following casts:

(int)
(string)
(pointer)

Thus, the call to obtain the time of day would look like the following:

```
int i;
i = (int) time(0);
```

Intrinsic functions are functions that are built into the **xgmon** program. A C programmer may extend the **xgmon** program and add new intrinsic functions.

Each intrinsic function is described using the following notation:

(*type*) *func_name* (*type param1*, *type param2*,...)

For example:

```
(int) fopen (string Filename, string Mode)
```

In this example, the `fopen` function returns a value of `int` type, and takes two parameters. Both parameters are of `string` type, the first being the file name and the second being the access mode.

The seven kinds of intrinsic functions are described below:

- Intrinsic Functions for Database Operations

  These intrinsic functions extract information from the internal database, and set and retrieve user-defined environment variables associated with hosts.

- Intrinsic Functions for Host Information

  These functions obtain information about hosts.

- Intrinsic Functions for Formatted Output

  These intrinsic functions provide the ability to create formatted text strings.

- Intrinsic Functions for Graphics Functions

  The graphics intrinsic functions permit virtual G machines to manipulate display elements on the topology display window and to treat the window associated with a virtual G machine as an all-points-addressable display.

- Intrinsic Functions for File Input/Output Operations

  These intrinsic functions are used in conjunction with the **print** statement to perform file input or output.

- Intrinsic Functions for Virtual G Machine Control

  These intrinsic functions provide information about the current virtual G machine environment and allow a virtual G machine to start programs in other virtual G machines or start up an AIX program.

- Intrinsic Functions for String Manipulation

  These intrinsic functions manipulate strings in a variety of ways.

Related Information

How to Install the AIX Network Management/6000 Licensed Program in *General Concepts and Procedures.*

xgmon Overview for Network Management in *Communication Concepts and Procedures.*

Alphabetic List of Intrinsic Functions

aix_exec Executes AIX programs and commands from within a VGM environment.

alloc Makes a specified amount of storage space available and returns a pointer to the newly allocated space.

ascii Returns the integer ASCII value of the first character in the specified string.

base_type Takes a Management Information Database (MIB) numeric-format variable name or numeric-format instance ID and returns a number that indicates its base type.

close Closes the open file indicated by the specified file descriptor.

ctime Generates a text string that corresponds to an integer expression of time.

dep_info Returns information about a display element.

dotaddr Returns a string representing the Internet Protocol (IP) address in dot notation.

draw_line Draws a line.

draw_string Enables the display of formatted output in color.

exec Allows a virtual G machine to start execution of a library command in another virtual machine or to issue a system command.

flush_trap Flushes the current trap being processed.

font_height Returns the height, in pixels, of the font being used in the graphics window associated with a virtual G machine (VGM).

font_width Returns the width, in pixels, of the font being used in the graphics window associated with a VGM.

fopen Opens the file indicated by the specified file name.

get_deps Returns a list of display elements that are grouped under a particular node.

getenv Obtains the value of a user-defined environment variable for a display element.

get_MIB_group
Finds the set of all Management Information Base (MIB) variable names that contain a given text string as a prefix.

get_primary Returns the current primary address associated with the specified host.

group_dep Maps a dynamically created node or host to the topology display window.

gw_var Extracts the value of the specified Management Information Base (MIB) numeric-format instance ID for the specified host from the internal database.

hexval Returns the integer value represented by the text characters in the specified string.

highlight_dep Permits a VGM to temporarily highlight a display element.

hostname Returns the text name of the host.

ipaddr Returns the normal, primary Internet Protocol (IP) address of the specified host.

left Extracts a substring beginning at the leftmost portion of the source string.

make_dep Dynamically creates a new node or host.

make_link Dynamically creates a link between two hosts.

mid Extracts a substring from within a source string.

move_dep Changes the relative location of a display element within a topology display window.

new_deps Returns a pointer to an array of strings representing the names of dynamically created display elements.

next_alternate Changes the current primary address of the specified host to the next available alternate address.

num Returns a string of text characters representing the decimal value of the specified integer.

password Returns the Simple Network Management Protocol (SNMP) community name associated with the specified host.

ping Sends an ICMP ECHO request to the specified host.

raise_window Attempts to raise the graphics window associated with the virtual G machine in which the program is running.

read Reads the next line in an open file specified by the file descriptor.

real_type Takes a Management Information Base (MIB) numeric-format variable name or numeric-format instance ID and returns a number indicating its actual MIB type.

rename_dep Renames a display element.

reuse_mem Controls garbage collection by a VGM.

right Extracts a substring from the rightmost portion of the source string.

set_element_mask
Allows a VGM to change the current display element mask.

setenv Sets the user-defined environment variable for a display element to the specified value.

snmp_var Returns the Management Information Base (MIB) numeric-format variable name associated with a specified MIB text-format variable name.

sprintf Enables formatted arguments.

strlen Returns the length of a string.

substr Searches a source string for a particular substring and returns the position of the leftmost occurrence of that substring.

time Returns the current system time.

val Returns the integer value represented by the text characters in the specified string.

window_height
Returns the height, in pixels, of the graphics window associated with a VGM.

window_width
Returns the width, in pixels, of the graphics window associated with a VGM.

words_free Returns the number of free words remaining in the data segment of the VGM.

Functional List of Intrinsic Functions

Database Operations

base_type Takes a Management Information Database (MIB) numeric-format variable name or numeric-format instance ID and returns a number that indicates its base type.

getenv Obtains the value of a user-defined environment variable for a display element.

get_MIB_group
Finds the set of all Management Information Base (MIB) variable names that contain a given text string as a prefix.

gw_var Extracts the value of the specified Management Information Base (MIB) numeric-format instance ID for the specified host from the internal database.

real_type Takes a Management Information Base (MIB) numeric-format variable name or numeric-format instance ID and returns a number indicating its actual MIB type.

setenv Sets the user-defined environment variable for a display element to the specified value.

snmp_var Returns the Management Information Base (MIB) numeric-format variable name associated with a specified MIB text-format variable name.

Host Information

dotaddr Returns a string representing the IP address in dot notation.

get_primary Returns the current primary address associated with the specified host.

hostname Returns the text name of the host.

ippaddr Returns the normal, primary Internet Protocol (IP) address of the specified host.

next_alternate
Changes the current primary address of the specified host to the next available alternate address.

password Returns the SNMP community name associated with the specified host.

ping Sends an ICMP ECHO request to the specified host.

String Manipulation

ascii Returns the integer ASCII value of the first character in the specified string.

hexval Returns the integer value represented by the text characters in the specified string.

left Extracts a substring beginning at the leftmost portion of the source string.

mid Extracts a substring from within a source string.

right Extracts a substring from the rightmost portion of the source string.

strlen Returns the length of a string.

substr Searches a source string for a particular substring and returns the position of the leftmost occurrence of that substring.

val Returns the integer value represented by the text characters in the specified string.

Formatted Output

num Returns a string of text characters representing the decimal value of the specified integer.

sprintf Enables formatted arguments.

File I/O

close Closes the open file indicated by the specified file descriptor.

fopen Opens the file indicated by the specified file name.

read Reads the next line in the open file specified by the file descriptor.

Virtual G Machine Control

aix_exec Executes AIX programs and commands from within a VGM environment.

alloc Makes a specified amount of storage space available and returns a pointer to the newly allocated space.

ctime Generates a text string that corresponds to an integer expression of time.

exec Allows a virtual G machine to start execution of a library command in another virtual G machine or to issue a system command.

flush_trap Flushes the current trap being processed.

reuse_mem Controls garbage collection by a VGM.

time Returns the current system time.

words_free Returns the number of free words remaining in the data segment of the VGM.

Graphics Functions

dep_info Returns information about a display element.

draw_line Draws a line.

draw_string Enables the display of formatted output in color.

font_height Returns the height, in pixels, of the font being used in the graphics window associated with a VGM.

font_width Returns the width, in pixels, of the font being used in the graphics window associated with a VGM.

get_deps Returns a list of display elements that are grouped under a particular node.

group_dep Maps a dynamically created node or host to the topology display window.

highlight_dep Permits a VGM to temporarily highlight a display element.

make_dep Dynamically creates a new node or host.

make_link Dynamically creates a link between two hosts.

move_dep Changes the relative location of a display element within a topology display window.

new_deps Returns a pointer to an array of strings representing the names of dynamically created display elements.

raise_window Attempts to raise the graphics window associated with the virtual G machine in which the program is running.

rename_dep Renames a display element.

set_element_mask
Allows a VGM to change the current display element mask.

window_height
Returns the height, in pixels, of the graphics window associated with a VGM.

window_width
Returns the width, in pixels, of the graphics window associated with a VGM.

How to Create xgmon Intrinsic Functions

Prerequisite Tasks or Conditions

Install the **xgmon** program.

Procedure

1. Go to your working directory.
2. Place a copy of the **/usr/lpp/xgmon/bin/xgmon.o** file and the **/usr/lpp/xgmon/bin/user_func.c** file in your working directory. If you have previously extended the **xgmon** program using files in addition to the **user_func.c** file, place a copy of those **.o** files in your working directory. For example, if new functions were previously implemented in the **/usr/lpp/xgmon/bin/new_code.c** file, place a copy of the **/usr/lpp/xgmon/bin/new_code.o** file in your working directory.

   The following is the template of a **user_func.c** file:

```
/* XGMON user-defined functions */

/* function types: integer, string, pointer */
#define T_INT 256
#define T_STRING 512
#define T_POINTER 1024

/* declare function names here */

/* User-defined functions are made available by adding an entry
to each of the following three tables.

user_func_name: add a string corresponding to the function name
user_func_type: add a bitmask corresponding to the types of the
                 values that can be returned by the function
user_func_rout: add the name of the routine to be called using
                 prototype:

                 routine(argc,argv)
                 int argc;     //argument count
                 unsigned long argv[]; //the arguments

*/

/* user function names */
char *user_func_name[] = {
     };

/* user function types */
int user_func_type[] = {
     };

/* user function routines */
unsigned long (*user_funct_rout[])() = {
     };
```

3. Choose a name for the new function, determine the data type(s) (integer, pointer, or string) the function will return, choose a name for the C routine, and write the C routine for the new function. Add the function name, return data type(s), and C routine name to the appropriate tables that are supplied in the **/usr/lpp/xgmon/bin/user_func.c** file. All entries must be made to the same relative location in all the respective tables. Proceed as follows:

   a. Add a string corresponding to the name of the intrinsic function to the **user_func_name** table.

      For example, if the name of the new intrinsic function is **cast**, enter:

      ```
      char *user_func_name[] = {
           "cast",
            ""
      };
      ```

      The end of the table is marked by a null string.

      Note: To override an existing intrinsic function, give the new function the same name as the function it replaces.

   b. Add a bitmask corresponding to the data types that can be returned by the function to the **user_func_type** table.

      For example, if the new intrinsic function is intended to return any possible type, specify the data type as the logical OR of the types `T_INT`, `T_STRING`, and `T_POINTER` by entering:

      ```
      int user_func_type[] = {
           T_INT | T_STRING | T_POINTER
      };
      ```

   c. Declare the name of the C routine as a function at the top of the **user_func.c** file. Add the name in the corresponding position in the **user_func_rout** table.

      For example, if the name of the C routine implementing the intrinsic function is **cast_proc**, enter the following at the top of the **user_func.c** file under the `/* declare function names here */` comment:

      ```
      static unsigned long cast_proc();
      ```

      Then, enter the following in the **user_func_rout** table:

      ```
      unsigned long (*user_funct_rout[])() = {
           cast_proc
      };
      ```

   d. Place the C routine for the new intrinsic function at the end of the **user_func.c** file:

      ```
      static unsigned long cast_proc(argc,argv)
      int argc;
      unsigned long argv[];
      {
          return(argv[0]);
      }
      ```

Note: Any user-defined intrinsic function is called similar to a C program. It has two parameters. The first is the count of the number of arguments the function is passed. The second is an array, each element of which is the value of the corresponding argument. The value the C routine returns will be the result of the intrinsic function as seen by the **xgmon** programming utility that is executing.

If desired, the C routine for the new intrinsic function can be placed in a different file; for example, the **new_code.c** file.

4. Create the **user_func.o** file by compiling the **user_func.c** file as follows:

```
cc -c user_func.c
```

If the C routine for the new intrinsic function is stored in a different file, compile that file as well. For example, if the function is stored in the **new_code.c** file, compile as follows:

```
cc -c new_code.c
```

5. Build the new **xgmon** program by linking the new **user_func.o** file with the **xgmon.o** file and any other **.o** files that are needed. For example, assuming that the new functions are implemented in the `new_code.c` file, create a new **xgmon** program as follows:

```
cc -o xgmon+ xgmon.o user_func.o new_code.o -l X11
```

6. Rename the new **xgmon** program to **xgmon+** and move it to the **/usr/lpp/xgmon/bin** directory.
7. Place a copy of the **user_func. c** and the **user_func.o** files in the **/usr/lpp/xgmon/bin** directory. If the new intrinsic function is stored in a different file, such as the **new_code.c** file, move that file and its **.o** file to the **/usr/lpp/xgmon/bin** directory.

Related Information

Extending **xgmon** Intrinsic Functions on page 6–23, Understanding Version Control in *Communication Concepts and Procedures*.

How to Create **xgmon** Library Commands on page 6–46, How to Modify Existing **xgmon** Library Commands on page 6–48.

How to Create xgmon Library Commands

Prerequisite Tasks or Conditions

1. Install the **xgmon** program.
2. Set the **GLIB** environment variable to the **/usr/lpp/xgmon/lib** directory as follows:

   ```
   GLIB=/usr/lpp/xgmon/lib
   export GLIB
   ```

3. Change the directory to **/usr/lpp/xgmon** as follows:

   ```
   cd /usr/lpp/xgmon
   ```

4. Set the **PATH** environment variable as follows:

   ```
   $PATH:/usr/lpp/xgmon/bin
   export PATH
   ```

5. Start the **xgmon** program by typing:

   ```
   xgmon
   ```

Procedure

1. Switch to another shell or X11 window, and change to your working directory. For example, type:

   ```
   cd /u/test
   ```

2. Choose a name for your new library command; for example, **new_pgm**. According to the naming convention for library programs, a **.g** qualifier must be appended to the library command name; for example, **new_pgm.g**.
3. Code the new library program using the **xgmon** programming utility.
4. Return to the **xgmon** console.
5. Compile the new library program with the **compile** system command. For example, type:

   ```
   compile /u/test/new_pgm.g
   ```

6. If there are any errors, return to the previous shell or X11 window, correct the problems, and recompile the new library program at the **xgmon** console.
7. If this new library program has no parameters, test the new library command using the **start** system command. If the functionality of the code is not correct, fix the problem(s), and recompile and test the new library program.

   If the new library program has parameters, execute the **halt** system command to unload the virtual G machine that the **compile** system command loaded with the new library program object code. In this case, you cannot test the functionality of the code unless the library program resides in the directory specified by the **GLIB** environment variable.

8. Switch to another shell or X11 window. Move the new library program to the directory specified by the **GLIB** environment variable. For example, if the **GLIB** environment variable is set to **/usr/lpp/xgmon/lib**, type:

   ```
   mv /u/test/new_pgm.g /usr/lpp/xgmon/lib/.
   ```

9. Return to the **xgmon** console. Execute the new library command by entering at the **xgmon** command prompt (>) the name of the library command and any required parameters; for example:

   ```
   new_pgm Parameter1 Parameter2 ... Parametern
   ```

10. If the functionality of the new code is not correct, continue debugging the library program from the directory specified by the **GLIB** environment variable. If your new library command has parameters, do not use the **compile** system command to test the new library program. Continue to execute it as specified in the previous step.

Related Information

The **compile** system command, **halt** system command, **start** system command.

The **mv** command.

How to Create **xgmon** Intrinsic Functions on page 6–43, How to Modify Existing **xgmon** Library Commands on page 6–48.

How to Modify Existing xgmon Library Commands

Prerequisite Tasks or Conditions

1. Install the **xgmon** program.
2. Set the **GLIB** environment variable to the **/usr/lpp/xgmon/lib** directory as follows:

   ```
   GLIB=/usr/lpp/xgmon/lib
   export GLIB
   ```

3. Change the directory to **/usr/lpp/xgmon** as follows:

   ```
   cd /usr/lpp/xgmon
   ```

4. Set the **PATH** environment variable as follows:

   ```
   $PATH:/usr/lpp/xgmon/bin
   export PATH
   ```

5. Start the **xgmon** program by typing:

   ```
   xgmon
   ```

Procedure

1. Switch to another shell or X11 window, and change to your working directory. For example, type:

   ```
   cd /u/test
   ```

2. Copy the library program (must be a **.g** file) to a **.bak** file, and move the **.g** file to your working directory. For example, if the program you wish to modify is **ping_all.g**, type:

   ```
   cp /usr/lpp/xgmon/lib/ping_all.g /usr/lpp/xgmon/lib/ping_all.bak
   ```

   ```
   mv /usr/lpp/xgmon/lib/ping_all.g /u/test/
   ```

3. Edit the **.g** file with a text editor and save your changes.
4. Return to the **xgmon** console.
5. Compile the modified library program with the **compile** system command. For example, type:

   ```
   compile /u/test/ping_all.g
   ```

6. If there are any errors, return to the previous shell or X11 window, correct the problems, and recompile the modified library program at the **xgmon** console.
7. If this modified library program has no parameters, test the modified library command using the **start** system command. If the functionality of the code is not correct, fix the problem(s), and recompile and test the modified library program.

   If the modified library program has parameters, execute the **halt** system command to unload the virtual G machine that the **compile** system command loaded with the modified library program object code. In this case, you cannot test the functionality of the code unless the library program resides in the directory specified by the **GLIB** environment variable.

8. Switch to the previous shell or X11 window. Move the modified library program to the directory specified by the **GLIB** environment variable. For example, if the **GLIB** environment variable is set to **/usr/lpp/xgmon/lib**, type:

   ```
   mv /u/test/ping_all.g /usr/lpp/xgmon/lib/
   ```

9. Return to the **xgmon** console. Execute the modified library command by entering at the **xgmon** command prompt (>) the name of the library command and any required parameters. For example, enter:

   ```
   ping_all
   ```

   or, if the modified library command has parameters:

   LibraryCommand Parameter1 Parameter2 ... Parametern

10. If the functionality of the modified code is not correct, continue debugging the library program from the directory specified by the **GLIB** environment variable. If your modified library command has parameters, do not use the **compile** system command to test the modified library program. Continue to execute it as specified in the previous step.

Related Information

The **compile** system command, **halt** system command, **start** system command.

The **cp** command, **mv** command.

How to Create **xgmon** Intrinsic Functions on page 6–43, How to Create **xgmon** Library Commands on page 6–46.

Chapter 7. Remote Procedure Call (RPC)

Remote Procedure Call (RPC) is a protocol that provides the high-level communications paradigm used in the operating system. RPC implements a logical client-to-server communications system designed specifically for the support of network applications. This chapter contains information on the messages, authentication, language, and protocol compiler for RPC. The chapter is divided into RPC concepts, lists of subroutines, and examples.

Remote Procedure Call (RPC) Overview

Remote Procedure Call (RPC) is a protocol that provides the high-level communications paradigm used in the operating system. RPC presumes the existence of a low-level transport protocol, such as Transmission Control Protocol/Internet Protocol (TCP/IP) or User Datagram Protocol (UDP/IP), for carrying the message data between communicating programs. RPC implements a logical client-to-server communications system designed specifically for the support of network applications.

The RPC protocol is built on top of the eXternal Data Representation (XDR) protocol, which is used to standardize the representation of data in remote communications. XDR converts the parameters and results of each RPC service provided.

The RPC protocol enables users to work with remote procedures as if the procedures were local. The remote procedure calls are defined through routines contained in the RPC protocol. Each call message is matched with a reply message. The RPC protocol is a message-passing protocol that implements other non-RPC protocols such as batching and broadcasting remote calls. The RPC protocol also supports callback procedures and the **select** subroutine on the server side.

A *client* is a computer or process that accesses the services or resources of another process or computer on the network. A *server* is a computer that provides services and resources, and that implements network services. Each network *service* is a collection of remote programs. A remote program implements remote procedures. The procedures, their parameters, and the results are all documented in the specific program's protocol.

RPC provides an authentication process that identifies the server and client to each other. RPC includes a slot for the authentication parameters on every remote procedure call so that the caller can identify itself to the server. The client package generates and returns authentication parameters. RPC supports various types of authentication such as the UNIX and Data Encryption Standard (DES) systems.

In RPC, each server supplies a program that is a set of procedures. The combination of a host address, a program number, and a procedure number specifies one remote service procedure. In the RPC model, the client makes a procedure call to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, and sends a reply back to the client. The procedure call then returns to the client.

RPC is divided into three layers: highest, intermediate, and lowest. The RPC interface is generally used to communicate between processes on different workstations in a network. However, RPC works just as well for communication between different processes on the same workstation.

The port mapper program maps RPC program and version numbers to a transport-specific port number. The port mapper program makes dynamic binding of remote programs possible.

To write network applications using RPC, programmers need a working knowledge of network theory. For most applications, understanding the RPC mechanisms usually hidden by the **rpcgen** protocol compiler is also helpful. However, **rpcgen** circumvents the need for understanding the details of RPC.

Related Information

The **rpcgen** command.

Alphabetical List of RPC Subroutines and Macros on page 7–41, Functional List of RPC Subroutines and Macros on page 7–44.

List of RPC Examples on page 7–49.

Understanding the RPC Features on page 7–29, Programming in RPC on page 7–20, Understanding the Port Mapper Program on page 7–17, Understanding RPC Authentication on page 7–10, Understanding the RPC Language on page 7–31, Understanding the RPC Message Protocol on page 7–5, Understanding the RPC Model on page 7–2, Understanding the rpcgen Protocol Compiler on page 7–37.

eXternal Data Representation (XDR) Overview for Programming on page 3–1.

Understanding Protocols for TCP/IP, User Datagram Protocol in *Communication Concepts and Procedures*.

Understanding the RPC Model

The remote procedure call (RPC) model is similar to the local procedure call model. In the local model, the caller places arguments to a procedure in a specified location such as a result register. Then, the caller transfers control to the procedure. The caller eventually regains control, extracts the results of the procedure, and continues execution.

RPC works in a similar manner, in that one thread of control winds logically through two processes: the caller process and the server process. First, the caller process sends a call message that includes the procedure parameters to the server process. Then, the caller process waits for a reply message (blocks). Next, a process on the server side, which is dormant until the arrival of the call message, extracts the procedure parameters, computes the results, and sends a reply message. The server waits for the next call message. Finally, a process on the caller receives the reply message, extracts the results of the procedure, and the caller resumes execution.

See the following figure for an illustration of the RPC Paradigm.

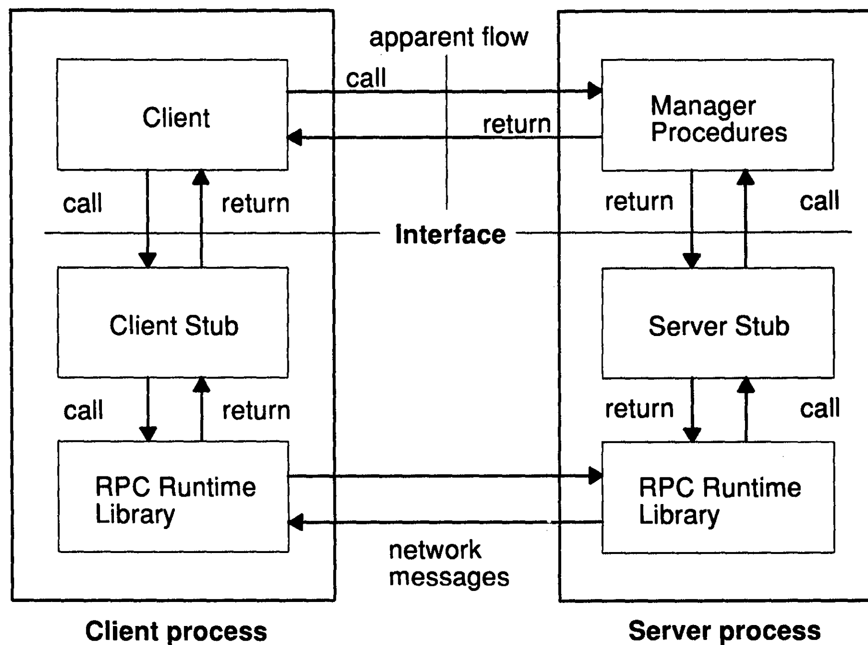


Figure 1. Remote Procedure Call Flow

In the RPC model, only one of the two processes is active at any given time. Furthermore, this model is only an example. The RPC protocol makes no restrictions on the concurrency model implemented and others are possible. For example, an implementation may choose asynchronous remote procedure calls so that the client can continue working while waiting for a reply from the server. Additionally, the server can create a task to process incoming requests and thereby remain free to receive other requests.

Transports and Semantics

The RPC protocol is independent of transport protocols. That is, RPC does not care how a message is passed from one process to another. The protocol deals only with the specification and interpretation of messages.

RPC does not try to implement any kind of reliability. The application must be aware of the type of transport protocol underneath RPC. If the application is running on top of a reliable transport, such as TCP/IP, then most of the work is already done. If the application is running on top of a less reliable transport, such as UDP/IP, then the application must implement a retransmission and time-out policy, because RPC does not provide these services.

Due to transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. The semantics can be inferred from (and should be explicitly specified by) the underlying transport protocol. For example, consider RPC running on top of a transport such as UDP/IP. If an application retransmits RPC messages after short time outs and receives no reply, then the application infers that the procedure was executed zero or more times. If the application receives a reply, then the application infers that the procedure was executed at least once.

A transaction ID is packaged with every RPC request. In order to ensure some degree of execute-at-most-once semantics, RPC allows a server to use the transaction ID to recall a previously granted request. The server can then refuse to grant that request again. The server is allowed to examine the transaction ID only as a test for equality. The RPC client mainly uses the transaction ID to match replies with requests. However, a client application can reuse a transaction ID when transmitting a request.

When using a reliable transport such as TCP/IP, the application can infer from a reply message that the procedure was executed exactly once. If the application receives no reply message, the application cannot assume that the remote procedure was not executed. Note that even if a connection-oriented protocol like TCP/IP is used, an application still needs time outs and reconnection to handle server crashes.

There are possibilities for transports besides datagram or connection-oriented protocols. For example, a request-reply protocol, such as Versatile Message Transaction Protocol (VMTP), is perhaps the most natural transport for RPC.

RPC in the Binding Process

The act of binding a client to a service is *not* part of the remote procedure call specification. This important and necessary function is left to higher level software. However, the higher level software may use RPC in the binding process. The Port Mapper Program is an example of software that uses RPC.

The RPC protocol's relationship to the binding software is similar to the relationship of the network jump-subroutine instruction (JSR) to the loader (binder). The loader uses JSR to accomplish its task. Likewise, the network uses RPC to accomplish the bind.

Related Information

List of RPC Examples on page 7–49.

Programming in RPC on page 7–20, Understanding the Port Mapper Program on page 7–17, Understanding the RPC Language on page 7–31, Understanding the RPC Message Protocol on page 7–5.

Understanding Protocols for TCP/IP, User Datagram Protocol in *Communication Concepts and Procedures*.

Understanding the RPC Message Protocol

The RPC message protocol consists of two distinct structures: the call message and the reply message. A client makes a remote procedure call to a network server and receives a reply containing the results of the procedure's execution. By providing a unique specification for the remote procedure, RPC can match a reply message to each call (or request) message.

The RPC message protocol is defined using the XDR data description, which includes structures, enumerations, and unions. See Using the RPC Language Descriptions on page 7–31 for more information.

When RPC messages are passed using the TCP/IP byte stream protocol for data transport, it is important to identify the end of one message and the start of the next one.

Understanding the RPC Protocol Requirements

The RPC message protocol requirements are:

- Unique specification of a procedure to call
- Matching of response messages to request messages
- Authentication of caller to service and service to caller.

In addition to these requirements, features that detect the following are worth supporting because of protocol roll-over errors, implementation bugs, user errors, and network administration:

- RPC protocol mismatches
- Remote program protocol version mismatches
- Protocol errors (such as misspecification of a procedure's parameters)
- Reasons why remote authentication failed
- Any other reasons why the desired procedure was not called.

Understanding the RPC Messages

The initial structure of an RPC message is as follows:

```
struct rpc_msg {
     unsigned int  xid;
     union switch (enum msg_type mtype)  {
          case CALL:
               call_body cbody;
          case REPLY;
               reply_body rbody;
     } body;
};
```

All remote procedure call and reply messages start with a transaction identifier, `xid`, which is followed by a two-armed discriminated union. The union's discriminant is `msg_type`, which switches to one of the following message types: `CALL` or `REPLY`. The `msg_type` has the following enumeration:

```
enum msg_type {
     CALL     = 0,
     REPLY    = 1
};
```

The xid parameter is used by clients matching a reply message to a call message or by servers detecting retransmissions. The server side does not treat the xid parameter as a sequence number.

The initial structure of an RPC message is followed by the body of the message. The body of a call message has one form. The body of a reply message takes one of two forms, depending on whether a call is accepted or rejected by the server.

Understanding an RPC Call Message

Each remote procedure call message contains the following unsigned integer fields to uniquely identify the remote procedure:

- Program number
- Program version number
- Procedure number.

The body of an RPC call message takes the following form:

```
struct call_body {
     unsigned int rpcvers;
     unsigned int prog;
     unsigned int vers;
     unsigned int proc;
     opaque_auth cred;
     opaque_auth verf;
     1 parameter
     2 parameter . . .
};
```

The parameters for the call message body structure are defined as follows:

rpcvers Specifies the version number of the RPC protocol. The value of this parameter is 2 for the second version of RPC.

prog Specifies the number that identifies the remote program. This is an assigned number represented in a protocol that identifies the program needed to call a remote procedure. Program numbers are administered by a central authority and documented in the program's protocol specification.

vers Specifies the number that identifies the remote program version. As a remote program's protocols are implemented, they evolve and change. Version numbers are assigned to identify different stages of a protocol's evolution. Servers can service requests for different versions of the same protocol simultaneously.

proc Specifies the number of the procedure associated with the remote program being called. These numbers are documented in the specific program's protocol specification. For example, a protocol's specification can list the read procedure as procedure number 5 or the write procedure as procedure number 12.

cred Specifies the credentials-authentication parameter that identifies the caller as having permission to call the remote program. This parameter is passed as an opaque data structure, which means the data is not interpreted as it is passed from the client to the server.

- *verf*
  Specifies the verifier-authentication parameter that identifies the caller to the server. This parameter is passed as an opaque data structure, which means the data is not interpreted as it is passed from the client to the server.

1 parameter Denotes a procedure-specific parameter.

2 parameter Denotes a procedure-specific parameter.

The client can send a broadcast packet to the network and wait for numerous replies from various servers. The client can also send an arbitrarily large sequence of call messages in a batch to the server.

Understanding an RPC Reply Message

The RPC protocol for a reply message varies depending on whether the call message is accepted or rejected by the network server.

The reply message to a request contains information to distinguish the following conditions:

- RPC executed the call message successfully.
- The remote implementation of RPC is not protocol version 2. The lowest and highest supported RPC version numbers are returned.
- The remote program is not available on the remote system.
- The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
- The requested procedure number does not exist. This is usually a caller-side protocol or programming error.

The RPC reply message takes the following form:

```
enum reply_stat stat {
     MSG_ACCEPTED = 0,
     MSG_DENIED   = 1
};
```

The `enum reply_stat` discriminant acts as a switch to the rejected or accepted reply message forms.

The Reply to an Accepted Request

The `body` of an RPC reply message for a request that is accepted by the network server has the following structure:

```
struct accepted_reply areply {
     opaque_auth verf;
     union switch (enum accept_stat stat) {
          case SUCCESS:
               opaque results {0};
               /* procedure specific results start here */
          case PROG_MISMATCH:
               struct {
                    unsigned int low;
                    unsigned int high;
               } mismatch_info;
          default:
               void;
     } reply_data;
};
```

The structures within the accepted reply are defined as follows:

`opaque_auth` *verf*
: Authentication verifier generated by the server to identify itself to the caller.

`enum accept_stat`
: A discriminant that acts as a switch between `SUCCESS`, `PROG_MISMATCH`, and other appropriate conditions which are defined as follows:

`SUCCESS`	RPC call is successful.
`PROG_UNAVAIL`	The remote server has not exported the program.
`PROG_MISMATCH`	The remote server cannot support the client's version number. Returns the lowest and highest version numbers of the remote program that are supported by the server.
`PROC_UNAVAIL`	The program cannot support the requested procedure.
`GARBAGE_ARGS`	The procedure cannot decode the parameters specified in the call.

The **accept_stat** enumeration data type has the following definitions:

```
enum accept_stat  {
     SUCCESS       = 0,   /* RPC executed successfully        */
     PROG_UNAVAIL  = 1,   /* remote has not exported program  */
     PROG_MISMATCH = 2,   /* remote cannot support version #  */
     PROC_UNAVAIL  = 3,   /* program cannot support procedure */
     GARBAGE_ARGS  = 4,   /* procedure cannot decode params   */
};
```

Note: An error condition may exist even when a call message is accepted by the server.

The Reply to a Rejected Request

A call message can be rejected by the server for two reasons: either the server is not running a compatible version of the RPC protocol, or there is an authentication failure.

The `body` of an RPC reply message for a request that is rejected by the network server has the following structure:

```
struct rejected_reply rreply {
union switch (enum reject_stat stat) {
     case RPC_MISMATCH:
          struct {
               unsigned int low;
               unsigned int high;
          } mismatch_info;
     case AUTH_ERROR:
          enum auth_stat stat;
};
```

The enum reject_stat discriminant acts as a switch between RPC_MISMATCH and AUTH_ERROR. The rejected call message returns one of the following status conditions:

```
enum reject_stat {
     RPC_MISMATCH    = 0, /* RPC version number is not 2       */
     AUTH_ERROR      = 1, /* remote cannot authenticate caller */
};
```

RPC_MISMATCH
: The server is not running a compatible version of the RPC protocol. The server returns the lowest and highest version numbers available.

AUTH_ERROR
: The server refuses to authenticate the caller and returns a failure status with the value enum auth_stat. Authentication may fail because of bad or rejected credentials, bad or rejected verifier, expired or replayed verifier, or security problems.

: If the server does not authenticate the caller, AUTH_ERROR returns one of the following conditions as the failure status:

```
enum auth_stat {
     AUTH_BADCRED        = 1, /* bad credentials              */
     AUTH_REJECTEDCRED   = 2, /* client must begin new session */
     AUTH_BADVERF        = 3, /* bad verifier                 */
     AUTH_REJECTEDVERF   = 4, /* verifier expired or replayed  */
     AUTH_TOOWEAK        = 5, /* rejected for security reasons */
};
```

Marking Records in RPC Messages

When RPC messages are passed using the TCP/IP byte stream protocol for data transport, it is important to identify the end of one message and the start of the next one. This is called Record Marking (RM).

A record is composed of one or more record fragments. A record fragment is a 4-byte header. The header is followed by 0 to 232–1 bytes of fragment data. The bytes encode an unsigned binary number, similar to XDR integers, in which the order of bytes is from highest to lowest. This binary number encodes a Boolean and an unsigned binary value of 31 bits.

The Boolean value is the highest-order bit of the header. If the Boolean value is 1 (one), the fragment is the last fragment of the record. The unsigned binary value is the length in bytes of the data fragment. If the parameters to the remote procedure appear as garbage to the server, it is usually caused by a protocol disagreement between client and service.

Related Information

List of RPC Examples on page 7–49.

Programming in RPC on page 7–20, Understanding RPC Authentication on page 7–10, Using the RPC Language Descriptions on page 7–31, Understanding the RPC Model on page 7–2.

eXternal Data Representation (XDR) Overview for Programming on page 3–1.

Understanding Protocols for TCP/IP, User Datagram Protocol in *Communication Concepts and Procedures.*

Understanding RPC Authentication

The caller may not want to identify itself to the server, and the server may not require an ID from the caller. However, some network services, such as the Network File System, require stronger security. RPC authentication provides a certain degree of security. The following are part of RPC authentication:

- Understanding RPC Authentication Protocol
- Understanding NULL Authentication
- Understanding UNIX Authentication
- Understanding Data Encryption Standard (DES) Authentication
- Understanding Data Encryption Standard (DES) Authentication Protocol
- Understanding Diffie-Hellman Encryption.

RPC deals only with authentication and not with access control of individual services. Each service must implement its own access control policy and reflect this policy as return statuses in its protocol. The programmer can build additional security and access controls on top of the message authentication.

The authentication subsystem of the RPC package is open ended. Different forms of authentication can be associated with RPC clients. That is, multiple types of authentication are easily supported at one time. Examples of authentication types include UNIX, DES, and NULL. The default authentication type is none (AUTH_NULL).

Understanding RPC Authentication Protocol

The RPC protocol provisions for authentication of the caller to the service, and vice versa, are provided as part of the RPC protocol. Every remote procedure call is authenticated by the RPC package on the server, and similarly, the RPC client package generates and sends authentication parameters. The call message has two authentication fields: credentials and verifier. The reply message has one authentication field: response verifier.

The RPC protocol specification defines the credentials of the call message and the verifiers of both the call message and the reply message as an opaque data type, as follows:

```
enum auth_flavor {
      AUTH_NULL     = 0,
     AUTH_UNIX     = 1,
     AUTH_SHORT    = 2,
     AUTH_DES      = 3
     /* and more to be defined */
};
struct opaque_auth {
     auth_flavor flavor;
     opaque body<400>;
};
```

Any `opaque_auth` structure is an `auth_flavor` enumeration followed by bytes which are opaque to the RPC protocol implementation. The interpretation and semantics of the data contained within the authentication fields are specified by individual, independent authentication protocol specifications.

If authentication parameters were rejected, the response message contains information stating why they were rejected. A server can support multiple types of authentication at one time. If authentication parameters are rejected, the response message contains information stating the reason.

Understanding NULL Authentication

Sometimes, the RPC caller does not know its own identity or the server does not need to know the caller's identity. In these cases, the AUTH_NULL authentication type can be used as the flavor in both the RPC call message and the response message. The bytes of the `opaque_auth` body are undefined. It is recommended that the opaque length be 0 (zero).

Understanding UNIX Authentication

The caller of a remote procedure may wish to identify himself as he is identified on the UNIX system. The value of the credential's discriminant of an RPC call message is AUTH_UNIX. The bytes of the credential's opaque body encode the following structure:

```
struct auth_unix {
     unsigned     stamp;
     string       machinename<255>;
     unsigned     uid;
     unsigned     gid;
     unsigned     gids<10>;
};
```

The parameters in the structure are defined as follows:

stamp	Specifies the arbitrary ID generated by the caller's workstation.
machinename<255>	Specifies the name of the caller's workstation. The name must not exceed 255 bytes in length.
uid	Specifies the caller's effective user ID.
gid	Specifies the caller's effective group ID.
gids<10>	Specifies the counted array of groups that contain the caller as a member. A maximum of 10 groups is allowed.

The verifier accompanying the credentials should be `AUTH_NULL`.

The value of the discriminant in the response verifier of the reply message from the server is either `AUTH_NULL` or `AUTH_SHORT`. If the value is `AUTH_SHORT`, the bytes of the response verifier's string encode an opaque structure. The new opaque structure can then be passed to the server in place of the original `AUTH_UNIX` credentials. The server maintains a cache that maps shorthand opaque structures (passed back by way of an `AUTH_SHORT style` response verifier) to the original credentials of the caller. The caller saves network bandwidth and server CPU time when the shorthand credentials are used.

Note: The server can eliminate, or flush, the shorthand opaque structures at any time. If this happens, the RPC message is rejected due to an `AUTH_REJECTEDCRED` authentication error. The original `AUTH_UNIX` credentials can be used when this happens.

UNIX Authentication on the Client Side

When a caller creates a new RPC client handle, the authentication handle of the appropriate transport is set to the default using the **authnone_create** subroutine. The default for an RPC authentication handle is NULL. After creating the client handle, the client can select UNIX authentication, using the **authunix_create** routine. This routine creates an authentication handle with AIX permissions and causes each remote procedure call associated with the handle to carry with it the UNIX credentials.

Authentication information can be destroyed with the **auth_destroy** subroutine. Authentication information should be destroyed if one is attempting to conserve memory.

For more information, see the Example Using UNIX Authentication on page 7–50.

UNIX Authentication on the Server Side

Dealing with authentication issues on the server side is more difficult than dealing with them on the client side. The caller's RPC package passes the service dispatch routine a request that has an arbitrary authentication style associated with it. The server must then determine which style of authentication the caller used and whether the style is supported by the RPC package.

If the authentication parameter type is not suitable for the calling service, the service dispatch routine calls the **svcerr_weakauth** routine to refuse the remote procedure call. It is *not* customary for the server to check the authentication parameters associated with procedure 0 (NULLPROC).

If the service does not have the requested protocol, the service dispatch returns a status for access denied. The **svcerr_systemerr** primitive is called to detect a system error that is not covered by a service protocol.

Understanding Data Encryption Standard (DES) Authentication

DES authentication offers more security features than UNIX authentication. In order for DES authentication to work, the **keyserv** daemon must be running on both the server and client machines. The users at these workstations need public keys assigned in the public key database by the person administering the network. Additionally, each user's secret key must be decrypted using their **keylogin** command password.

DES authentication can handle the following UNIX problems:

- The naming scheme within UNIX authentication is UNIX-system oriented.
- UNIX authentication lacks a verifier, thereby allowing falsification of credentials.

For more information, see the Example Using DES Authentication on page 7–53.

DES Authentication Naming Scheme

DES addresses the caller with a simple string of characters instead of an operating system-specific integer. This string of characters is known as the caller's network name, or *netname*. The server is allowed to interpret the contents of the netname only for the purpose of identification of the caller. Therefore, netnames should be unique for each caller in the network.

Each operating system is responsible for implementing DES authentication to generate unique netnames for calling on remote servers. Operating systems can already distinguish local users to their systems, so extending this mechanism to the network is simple.

For example, a UNIX user at IBM with a user ID of 515 might be assigned the following netname: `unix.515@ibm.com`. This netname contains three items that serve to insure it is unique. Going backwards, there is only one naming domain called `ibm.com` in the internet. Within this domain, there is only one UNIX user with user ID 515. However, there may be another user on another operating system, for example VMS, within the same naming domain that, by coincidence, happens to have the same user ID. To insure that these two users can be distinguished we add the operating system name. So one user is `unix.515@ibm.com` and the other is `vms.515@ibm.com`.

The first field is actually a naming method rather than an operating system name. However, there is currently a one-to-one correspondence between naming methods and operating systems. If a naming standard is universally agreed upon in the future, the first field will become the name of that standard, instead of an operating system name.

DES Authentication Verifiers

Unlike UNIX authentication, DES authentication has a verifier that permits the server to validate the client's credential and the client to validate the server's credential. The contents of this verifier is primarily an encrypted timestamp. The timestamp is encrypted by the client and decrypted by the server. If the timestamp is close to the real time, then the client encrypted it correctly. In order to encrypt the timestamp correctly, the client must have the *conversation key* of the RPC session. The client with the conversation key is the authentic client.

The conversation key is a DES key that the client generates and includes in its first remote procedure call to the server. The conversation key is encrypted using a public key scheme in the first transaction. The particular public key scheme used in DES authentication is Diffie-Hellman with 192-bit keys. For more information, see the section on Understanding Diffie-Hellman Encryption) on page 7–15.

For successful validation, the client and the server need the same notion of the current time. If network time synchronization cannot be guaranteed, the client can synchronize with the server before beginning the conversation, perhaps by consulting the Internet Time Server (TIME).

DES Authentication on the Server Side

Determining the validity of a client's timestamp depends on whether it is the first transaction. In the first transaction, the server checks only that the timestamp has not expired. For all transactions other than the first transaction, the server verifies that the timestamp is greater than the previous timestamp from the same client, and that the timestamp has not expired. A timestamp has expired if the server's time is later than the sum of the client's timestamp plus the client's window. The sum of the timestamp plus the client window can be thought of as the lifetime of the credential.

DES Authentication on the Client Side

In the first transaction to the server, the client sends an encrypted item, the *window verifier*, that must be equal to the client's window minus one as an added check. Otherwise, it would be easy for the client to send random data in place of the timestamp with a fairly good chance of succeeding. Any other value for the credential is rejected by the server. If the window verifier is accepted by the server, the server returns to the client a verifier equal to the encrypted timestamp minus one second. If the client receives a different timestamp from the server, the client rejects it.

For all subsequent transactions, the client's timestamp is valid if it is greater than the previous timestamp and has not expired. A timestamp has expired if the server's time is later than the sum of the client's timestamp plus the client's window. The sum of the timestamp plus the client window can be thought of as the lifetime of the credential.

To use DES authentication, the programmer must set the client authentication handle using the **authdes_create** subroutine. This routine requires the network name of the owner of the server process, a lifetime for the credential, the address of the host with which to synchronize, and the address of a DES encryption key to use for encrypting timestamps and data.

Nicknames

The server's DES authentication subsystem returns a *nickname* to the client in the verifier response to the first transaction. The nickname is an unsigned integer. The nickname is likely to be an index into a table on the server that stores each client's netname, decrypted DES key, and window. The client can use the nickname in all subsequent transactions instead of passing its netname, encrypted DES key, and window each time. Using the nickname is not required, but its use can save time.

Clock Synchronization

Although the client and server clocks are originally synchronized, they can lose this synchronization. When this happens the client RPC subsystem normally receives the RPC_AUTHERROR error message and should resynchronize.

A client can also receive the RPC_AUTHERROR message, even when the clocks are synchronized. This is because the server's nickname table has been flushed due to size limitations of the table or during a server crash. To receive new nicknames, all clients must re-send their original credentials to the server.

Understanding Data Encryption Standard (DES) Authentication Protocol

DES authentication has the following form of XDR enumeration:

```
enum authdes_namekind {
     ADN_FULLNAME = 0,
     ADN_NICKNAME = 1
};
typedef opaque des_block[8];
const MAXNETNAMELEN = 255;
```

A credential is either a client's full network name or its nickname. In the first transaction with the server, the client must use its full name. In all further transactions with the server the client can use its nickname. DES authentication protocol includes a 64-bit block of encrypted DES data and specifies the maximum length of a network user's name.

The `authdes_cred` union provides a switch between the full name and nickname forms, as follows:

```
union authdes_cred switch (authdes_namekind adc_namekind) {
     case ADN_FULLNAME:
          authdes_fullname adc_fullname;
     case ADN_NICKNAME:
          unsigned int adc_nickname;
};
```

The full name contains the network name of the client, an encrypted conversation key, and the window. The window is actually a lifetime for the credential. The server can terminate a client's timestamp and not grant the request if the time indicated by the verifier timestamp plus the window has expired. In the first transaction, the server confirms that the window verifier is one second less than the window. To ensure that requests are granted only once, the server can require timestamps in subsequent requests to be greater than the client's previous timestamps.

The structure for a credential using the client's full network name follows:

```
struct authdes_fullname {
    string name<MAXNETNAMELEN>; /* name of client              */
    des_block key;              /* PK encrypted conversation key*/
    unsigned int window;        /* encrypted window             */
};
```

A timestamp encodes the time since midnight, January 1, 1970. The structure for the timestamp follows:

```
struct timestamp {
     unsigned int seconds;       /* seconds             */
     unsigned int useconds;      /* and microseconds    */
```

The client verifier has the following structure:

```
struct {
     adv_timestamp;            /* one DES block          */
     adc_fullname.window;      /* one half DES block     */
     adv_winverf;              /* one half DES block     */
}
```

The window verifier is only used in the first transaction. In conjunction with the `fullname` credential, these items are packed into the above structure before being encrypted.

This structure is encrypted using CBC mode encryption with an input vector of zero. All other encryptions of timestamps use ECB mode encryption. The client's verifier has the following structure:

```
struct authdes_verf_clnt {
     timestamp adv_timestamp;      /* encrypted timestamp        */
     unsigned int adv_winverf;     /* encrypted window verifier  */
};
```

The server returns the client's timestamp minus one second in an encrypted response verifier. This verifier also sends the client an unencrypted nickname to be used in future transactions. The verifier from the server has the following structure:

```
struct authdes_verf_svr {
     timestamp adv_timeverf;      /* encrypted verifier     */
     unsigned int adv_nickname;   /* new nickname for client */
};
```

Understanding Diffie-Hellman Encryption

The particular public key scheme used in DES authentication is Diffie-Hellman with 192-bit keys. In the Diffie-Hellman encryption scheme, there are two constants, `BASE` and `MODULUS`. The particular values chosen for these for the DES authentication protocol are:

```
const BASE = 3;
const MODULUS = "d4a0ba0250b6fd2ec626e7efd637df76c716e22d0944b88b";
/* hex */
```

For example, two programmers A and B can send encrypted messages to each other in the following manner. First, programmers A and B independently generate secret keys at random, which can be represented as SK(A) and SK(B). Both programmers then publish their public keys PK(A) and PK(B) in a public directory. These public keys are computed from the secret keys as follows:

```
PK(A) = ( BASE ** SK(A) ) mod MODULUS
PK(B) = ( BASE ** SK(B) ) mod MODULUS
```

The ** (double asterisk) notation is used here to represent exponentiation. Now, both programmers A and B can arrive at the common key between them, represented here as CK(A, B), without revealing their secret keys.

Programmer A computes:

```
CK(A, B) = ( PK(B) ** SK(A)) mod MODULUS
```

while programmer B computes:

```
CK(A, B) = ( PK(A) ** SK(B)) mod MODULUS
```

These two can be shown to be equivalent:

```
(PK(B) ** SK(A)) mod MODULUS = (PK(A) ** SK(B)) mod MODULUS
```

If the `mod MODULUS` parameter is omitted, modulo arithmetic can simplify things as follows:

```
PK(B) ** SK(A) = PK(A) ** SK(B)
```

Then, if the result of the previous computation on B replaces PK(B) and the previous computation of A replaces PK(A), the equation is:

```
((BASE ** SK(B)) ** SK(A) = (BASE ** SK(A)) ** SK(B)
```

This equation can be simplified as follows:

```
BASE ** (SK(A) * SK(B)) = BASE ** (SK(A) * SK(B))
```

This produces a common key CK(A,B). This common key is not used directly to encrypt the timestamps used in the protocol. Instead, it is used to encrypt a conversation key that is then used to encrypt the timestamps. This allows the common key to be used as little as possible to prevent it from being broken. Breaking the conversation key usually has less serious consequences because conversations are relatively short-lived.

The conversation key is encrypted using 56-bit DES keys, yet the common key is 192 bits. To reduce the number of bits, 56 bits are selected from the common key as follows. The middle-most 8-bytes are selected from the common key, and then parity is added to the lower order bit of each byte, producing a 56-bit key with 8 bits of parity.

Related Information

The **keylogin** command.

The **keyserv** daemon.

The **authdes_create** subroutine, **auth_destroy** subroutine, **authnone_create** subroutine, **authunix_create** subroutine, **svcerr_weakauth** subroutine, **svcerr_systemerr** subroutine.

List of RPC Examples on page 7–49.

Example Using DES Authentication on page 7–53, Example Using UNIX Authentication on page 7–50.

eXternal Data Representation (XDR) Overview for Programming on page 3–1.

Network File System (NFS) Overview for System Management in *Communication Concepts and Procedures*.

Understanding the RPC Port Mapper Program

Client programs must find the *port* numbers of the server programs that they intend to use. Network transports do not provide such a service; they merely provide process-to-process message transfer across a network. A message typically contains a transport address which contains a network number, a host number, and a port number.

A port is a logical communications channel in a host. A server process receives messages from the network by waiting on a port. How a process waits on a port varies from one operating system to another, but all systems provide mechanisms that suspend processes until a message arrives at a port. Therefore, messages are sent to the ports at which receiving processes wait for messages.

Ports allow message receivers to be specified in a way that is independent of the conventions of the receiving operating system. The portmapper protocol defines a network service that permits clients to look up the port number of any remote program supported by the server. Since the port mapper program can be implemented on any transport that provides the equivalent of ports, it works for all clients, all servers, and all networks.

The port mapper program maps RPC program and version numbers to transport-specific port numbers. The port mapper program makes dynamic binding of remote programs possible. This is desirable because the range of reserved port numbers is small and the number of potential remote programs large. When running only the port mapper on a reserved port, the port numbers of other remote programs can be determined by querying the port mapper.

The port mapper also aids in broadcast RPC. A given RPC program usually has different port number bindings on different machines, so there is no way to directly broadcast to all of these programs. The port mapper, however, has a fixed port number. To broadcast to a given program, the client sends its message to the port mapper located at the broadcast address. Each port mapper that picks up the broadcast then calls the local service specified by the client. When the port mapper receives a reply from the local service, it sends the reply back to the client.

Registering Ports

Every port mapper on every host is associated with port number 111. The port mapper is the only network service that must have a dedicated port. Other network services can be assigned port numbers statically or dynamically, as long as the services register their ports with their host's port mapper. For example, a server program based on an RPC library typically gets a port number at run time by calling an RPC library procedure. The programmer should note that a given network service can be associated with port number 256 on one server and with port number 885 on another, because a service on a given host can be associated with a different port every time its server program is started.

The delegation of port-to-remote program mapping to a port mapper also automates port number administration. Statically mapping ports and remote programs in a file duplicated on each client requires updating all mapping files whenever a new remote program is introduced to a network. The alternative of placing the port-to-program mappings in a shared NFS file would be too centralized, and if the file server were to go down, the whole network would go down with it.

The port-to-program mappings, which are maintained by the port mapper server, are called a *portmap*. The port mapper is started automatically whenever a machine is booted. Both the server programs and the client programs call port mapper procedures. As part of its initialization, a server program calls its host's port mapper to create a portmap entry.

Whereas server programs call port mapper programs to update portmap entries, clients call port mapper programs to query portmap entries. To find a remote program's port, a client sends an RPC call message to a server's portmapper. If the remote program is supported on the server, the port mapper returns the relevant port number in an RPC reply message. The client program can then send RPC call messages to the remote program's port. A client program can minimize port mapper calls by caching the port numbers of recently called remote programs.

Note: The port mapper provides an inherently stateful service because a port map is a set of associations between registrants and ports.

Understanding Port Mapper Protocol

The following is the port mapper protocol specification in RPC language:

```
const PMAP_PORT = 111;      /* port mapper port number      */
```

The mapping of program, version, and protocol to the port number is shown by the following structure:

```
struct mapping {
     unsigned int prog;
     unsigned int vers;
     unsigned int prot;
     unsigned int port;
};
```

The values supported for the `prot` parameter are:

```
const IPPROTO_TCP = 6;      /* protocol number for TCP/IP     */
const IPPROTO_UDP = 17;     /* protocol number for UDP/IP     */
```

The list of mappings takes the following structure:

```
struct *pmaplist {
     mapping map;
     pmaplist next;
};
```

The structure for arguments to the `callit` parameter follows:

```
struct call_args {
     unsigned int prog;
     unsigned int vers;
     unsigned int proc;
     opaque args<>;
};
```

The results of the `callit` parameter have the following structure:

```
struct call_result {
     unsigned int port;
     opaque res<>;
};
```

The structure for port mapper procedures follows:

```
program PMAP_PROG {
     version PMAP_VERS {
          void
          PMAPPROC_NULL(void)          = 0;
```

```
        bool
        PMAPPROC_SET(mapping)             = 1;

        bool
        PMAPPROC_UNSET(mapping)           = 2;

        unsigned int
        PMAPPROC_GETPORT(mapping)         = 3;

        pmaplist
        PMAPPROC_DUMP(void)               = 4;

             call_result
             PMAPPROC_CALLIT(call_args)   = 5;
        } = 2;
} = 100000;
```

Understanding Port Mapper Procedures

The port mapper program currently supports two protocols: UDP/IP and TCP/IP. The port mapper is contacted by port number 111 on either of these protocols.

A description of the port mapper procedures follows:

PMAPPROC_NULL This procedure does no work. By convention, procedure 0 (zero) of any protocol takes no parameters and returns no results.

PMAPPROC_SET When a program first becomes available on a machine, it registers itself with the port mapper program on the same machine. The program passes its program number (*prog)*, version number (*vers)*, transport protocol number (*prot)*, and the port (*port)* on which it awaits service request. The procedure returns a Boolean response whose value is TRUE if the procedure successfully established the mapping or FALSE otherwise. The procedure does not establish a mapping if one already exists for the tuple (*prog*, *vers*, *prot*).

PMAPPROC_UNSET When a program becomes unavailable, it should unregister itself with the port mapper program on the same machine. The parameters and results have meanings identical to those of the PMAPPROC_SET procedure. The protocol and port number fields of the argument are ignored.

PMAPPROC_GETPORT Given a program number (*prog)*, version number (*vers)*, and transport protocol number (*prot)*, this procedure returns the port number on which the program is awaiting call requests. A port value of 0 (zero) means the program has not been registered. The *port* parameter of the argument is then ignored.

PMAPPROC_DUMP This procedure enumerates all entries in the port mapper data base. The procedure takes no parameters and returns a list of program, version, protocol, and port values.

PMAPPROC_CALLIT This procedure allows a caller to call another remote procedure on the same machine without knowing the remote procedure's port number. It supports broadcasts to arbitrary remote programs through the well-known port mapper port. The *prog*, *vers*, and *proc* parameters, and the bytes of the *args* parameter of a remote procedure call represent the program number, version number, procedure number, and arguments, respectively. The

PMAPPROC_CALLIT procedure sends a response only if the procedure is successfully executed. The port mapper communicates with the remote program using UDP/IP only. The procedure returns the remote program's port number, and the bytes of results are the results of the remote procedure.

Related Information

List of RPC Examples on page 7–49.

Broadcasting Remote Procedure Calls on page 7–30.

Network File System (NFS) Overview for System Management, Understanding Protocols for TCP/IP, User Datagram Protocol in *Communication Concepts and Procedures.*

Programming in RPC

Remote procedure calls can be made from any language. RPC is generally used to communicate between processes on different workstations. However, RPC works just as well for communication between different processes on the same workstation.

The RPC interface can be seen as being divided into three layers: highest, intermediate, and lowest. The highest layer of RPC is totally transparent to the operating system, workstation, and network upon which it runs. This level is actually a method for using RPC routines, rather than a part of RPC proper. The intermediate layer of RPC is RPC proper. At the intermediate layer, the programmer need not consider details about sockets or other low-level implementation mechanisms. The programmer simply makes remote procedure calls to routines on other workstations. The lowest layer of RPC allows the programmer greatest control. Programs written at this level can be more efficient.

Intermediate and lower level RPC programming includes assigning program numbers, version numbers, and procedure numbers. An RPC server can be started from the **inetd** daemon.

Assigning Program Numbers

A central system authority administers the program number (*prog* parameter). A program number permits the implementation of a remote program. The first implementation of a program is usually version number 1.

A program number is assigned by groups of 0x20000000 (decimal 536870912), according to the following list:

0–1xxxxxxx This group of numbers is predefined and administered by the AIX system. The numbers should be identical for all system customers.

20000000–3xxxxxxx The user defines this group of numbers. The numbers are used for new applications and for debugging new programs.

40000000–5xxxxxxx This group of numbers is transient and is used for applications that generate program numbers dynamically.

60000000–7xxxxxxx Reserved.

80000000–9xxxxxxx	Reserved.
a0000000–bxxxxxxx	Reserved.
c0000000–dxxxxxxx	Reserved.
e0000000–fxxxxxxxx	Reserved.

The first group of numbers is predefined, and should be identical for all customers. If a customer develops an application that might be of general interest, that application can be registered by assigning a number in the first range. The second group of numbers is reserved for specific customer applications. This range is intended primarily for debugging new programs. The third group is reserved for applications that generate program numbers dynamically. The final groups are reserved for future use, and should not be used.

Assigning Version Numbers

Most new protocols evolve into more efficient, stable, and mature protocols. As a program evolves, a new version number (*vers* parameter) is assigned. The version number identifies which version of the protocol the caller is using. The first implementation of a remote program is usually designated as version number 1 (or a similar form). Version numbers make it possible to use old and new protocols through the same server. See the Example Using Multiple Program Versions on page 7–73.

Just as remote program protocols may change over several versions, the actual RPC message protocol can also change. Therefore, the call message also contains the RPC version number. In the second version of the RPC protocol specification, the version number is always 2.

Assigning Procedure Numbers

The procedure number (*proc* parameter) identifies the procedure to be called. The procedure number is documented in each program's protocol specification. For example, a file service protocol specification can list the read procedure as procedure 5 and the write procedure as procedure 12.

Using Registered RPC Programs

The RPC program numbers and protocol specifications of standard RPC services are in the header files in the **/usr/include/rpcsvc** directory. The RPC **/etc/rpc** file describes the RPC program numbers in text so that users can identify the number with the name. The names identified in the text can be used in place of RPC program numbers. These programs, however, constitute only a small subset of those that have been registered.

The following is a list of registered RPC programs including the program number, program name, and program description:

Number	Name	Description
100000	PMAPPROG	port mapper
100001	RSTATPROG	remote stats
100002	RUSERSPROG	remote users
100003	NFSPROG	nfs
100004	YPPROG	network information service (NIS)
100005	MOUNTPROG	mount demon

Number	Name	Description
100006	DBXPROG	remote dbx
100007	YPBINDPROG	yp binder
100008	WALLPROG	shutdown msg
100009	YPPASSWDPROG	yppasswd server
100010	ETHERSTATPROG	ether stats
100011	RQUOTAPROG	disk quotas
100012	SPRAYPROG	spray packets
100013	IBM3270PROG	3270 mapper
100014	IBMRJEPROG	RJE mapper
100015	SELNSVCPROG	selection service
100016	RDATABASEPROG	remote database access
100017	REXECPROG	remote execution
100018	ALICEPROG	Alice Office Automation
100019	SCHEDPROG	scheduling service
100020	LOCKPROG	local lock manager
100021	NETLOCKPROG	network lock manager
100022	X25PROG	x.25 inr protocol
100023	STATMON1PROG	status monitor 1
100024	STATMON2PROG	status monitor 2
100025	SELNLIBPROG	selection library
100026	BOOTPARAMPROG	boot parameters service
100027	MAZEPROG	mazewars game
100028	YPUPDATEPROG	yp update
100029	KEYSERVEPROG	key server
100030	SECURECMDPROG	secure login
100031	NETFWDIPROG	nfs net forwarder init
100032	NETFWDTPROG	nfs net forwarder trans
100033	SUNLINKMAP_PROG	sunlink MAP
100034	NETMONPROG	network monitor
100035	DBASEPROG	lightweight database
100036	PWDAUTHPROG	password authorization
100037	TFSPROG	translucent file svc

Number	Name	Description
100038	NSEPROG	nse server
100039	NSE_ACTIVATE_PROG	nse activate daemon
150001	PCNFSDPROG	pc passwd authorization
200000	PYRAMIDLOCKINGPROG	Pyramid-locking
200001	PYRAMIDSYS5	Pyramid-sys5
200002	CADDS_IMAGE	CV cadds_image
300001	ADT_RFLOCKPROG	ADT file locking.

Programmers who write remote procedure calls should make the highest layer of RPC available to other users by way of a simple C language front-end routine that entirely hides the networking. To illustrate a call at the highest level, a program can simply call the **rnusers** routine, a C routine that returns the number of users on a remote workstation. The user need not be explicitly aware of using RPC.

Other RPC service library routines available to the C programmer are:

rusers Returns information about users on a remote workstation.

havedisk Determines whether the remote workstation has a disk.

rstat Gets performance data from a remote kernel.

rwall Writes to a specified remote workstation.

yppasswd Updates a user password in the network information service (NIS).

RPC services, such as the **mount** and **spray** commands, are not available to the C programmer as service library routines. Though unavailable, these services have RPC program numbers and can be invoked with the **callrpc** subroutine. Most of these services have compilable **rpcgen** protocol description files that simplify the process of developing network applications.

For more information, see the Example Using the Highest Layer of RPC on page 7–58.

Using the Intermediate Layer of RPC

The intermediate layer RPC routines are used for most applications. The intermediate layer is sometimes overlooked in programming due to its simplicity and lack of flexibility. At this level, RPC does not allow time-out specifications, choice of transport, or process control in case of errors. Nor does the intermediate layer of RPC support multiple types of call authentication. The programmer often needs these kinds of control.

Remote procedure calls are made with the **registerrpc**, **callrpc**, and **svc_run** system routines, which belong to the intermediate layer of RPC. The first two routines, **registerrpc** and **callrpc**, are the most fundamental. The **registerrpc** routine obtains a unique system-wide procedure identification number. The **callrpc** routine executes the remote procedure call.

Each RPC procedure is uniquely defined by a program number, version number, and procedure number. The program number specifies a group of related remote procedures, each of which has a different procedure number. Each program also has a version number.

Therefore, when a minor change, such as adding a new procedure, is made to a remote service, a new program number need not be assigned.

The RPC interface also handles arbitrary data structures, regardless of the different byte orders or structure layout conventions at various workstations. For more information, see the Example Using the Intermediate Layer of RPC on page 7–59.

Using the registerrpc Routine

Only the User Datagram Protocol (UDP) transport mechanism can use the **registerrpc** routine. This routine is always safe in conjunction with calls generated by the **callrpc** routine. The UDP transport mechanism can deal only with arguments and results that are less than 8K bytes in length.

The RPC **registerrpc** routine includes the following parameters:

- Program number
- Version number
- Procedure number to be called
- procedure name
- XDR subroutine that decodes the procedure parameters
- XDR subroutine that encodes the procedure calls.

After registering the local procedure, the server program's main procedure calls the **svc_run** routine, which is the RPC library's remote procedure dispatcher. The **svc_run** routine then calls the remote procedure in response to RPC messages. The dispatcher uses the XDR data filters that are specified when the remote procedure is registered to handle decoding procedure arguments and encoding results.

Using the callrpc Routine

The RPC **callrpc** routine executes remote procedure calls. See the Example Using the Intermediate Layer of RPC on page 7–59.

The **callrpc** routine includes the following parameters:

- Name of the remote server workstation
- Program number
- Version number of the program
- Procedure number
- Input XDR filter primitive
- Argument to be encoded and passed to the remote procedure
- Output XDR filter for decoding the results returned by the remote procedure
- Pointer to the location where the procedure's results are to be stored.

Multiple arguments and results can be embedded in structures. If the **callrpc** routine completes successfully, it returns a value of 0 (zero). Otherwise, it returns a nonzero value. The return codes are cast into values of integer data type in the **<rpc/clnt.h>** header file.

If the **callrpc** routine gets no answer after several attempts to deliver a message, it returns with an error code. The delivery mechanism is User Datagram Protocol (UDP). Adjusting the number of retries or using a different protocol requires the use of the lower layer of the RPC library.

Passing Arbitrary Data Types

The RPC interface can handle arbitrary data structures, regardless of the different byte orders or structure layout conventions on different machines, by converting the structures to a network standard called eXternal Data Representation (XDR) before sending them over the wire. The process of converting from a particular machine representation to XDR format is called *serializing*, and the reverse process is called *deserializing*.

The input and output parameters of the **callrpc** and **registerrpc** routines can be a built-in or user-supplied procedure. For more information, see the Example Showing How RPC Passes Arbitrary Data Types on page 7–61.

The XDR language has the following built-in type routines:

- **xdr_bool**
- **xdr_char**
- **xdr_u_char**
- **xdr_enum**
- **xdr_int**
- **xdr_u_int**
- **xdr_long**
- **xdr_u_long**
- **xdr_short**
- **xdr_u_short**
- **xdr_wrapstring**

Although the **xdr_string** routine exists, it passes three parameters to its XDR routine and cannot be used with the **callrpc** and **registerrpc** routines, which pass only two parameters. However, the **xdr_string** routine can be called with the **xdr_wrapstring** routine, which also has only two parameters.

If completion is successful, XDR routines return a nonzero value (or TRUE, in the C language). Otherwise, XDR returns a value of 0 (zero, or FALSE).

In addition to the built-in primitives are the following prefabricated building blocks:

- **xdr_array**
- **xdr_bytes**
- **xdr_opaque**
- **xdr_pointer**
- **xdr_reference**
- **xdr_string**
- **xdr_union**
- **xdr_vector**

For the higher layers, RPC takes care of many details automatically. However, the lowest layer of the RPC library allows the programmer to change the default values for these details. The lowest layer of RPC requires familiarity with sockets and their system calls. For more information, see the Example Using the Lowest Layer of RPC on page 7–63 and the Example Using Multiple Program Versions on page 7–73.

The lowest layer of RPC may be necessary in the following situations:

- The programmer needs to use TCP/IP. Higher layers use UDP, which restricts RPC calls to 8K bytes of data. TCP/IP permits calls to send long streams of data.

- The programmer wants to allocate and free memory while serializing or deserializing messages with XDR routines. No system call at the higher levels explicitly permits freeing memory. XDR routines are used for memory allocation as well as for input and output.
- The programmer needs to perform authentication on the client or server side by supplying credentials or verifying them.

Allocating Memory with XDR

XDR routines not only do input and output, they also do memory allocation. Consider the following XDR routine `xdr_chararr1()`, which deals with a fixed array of bytes with length `SIZE`.

```
xdr_chararr1 (xdrsp, chararr)
     XDR *xdrsp;
     char chararr[];
{
      char *p;
      int len;

      p = chararr;
      len = SIZE;
      return (xdr_bytes (xdrsp, &p, &len, SIZE));
}
```

If space has already been allocated in `chararr`, it can be called from a server. For example:

```
char chararr [SIZE};
svc_getargs (transp, xdr_chararr1, chararr);
```

If you want XDR to do the allocation, you need to rewrite this routine in the following way:

```
xdr_chararr2 (xdrsp, chararrp)
     XDR *xdrsp;
     char **chararrp;
{
      int len;

      len = SIZE;
      return (xdr_bytes (xdrsp, charrarrp, &len, SIZE));
}
```

Then the RPC call might look like this:

```
char *arrptr;

arrptr = NULL;
svc_getargs (transp, xdr_chararr2, &arrptr);
/*
*Use the result here
*/
svc_freeargs (transp, xdr_chararr2, &arrptr);
```

The character array can be freed with the `svc_freeargs()` macro. This does not attempt to free any memory in the variable indicating it is NULL.

Each XDR routine is responsible for serializing, deserializing, and freeing memory. When an XDR routine is called from the **callrpc** routine, the serializing part is used. When called from the **svc_getargs** routine, the deserializer is used and when called from the **svc_freeargs** routine, the memory deallocator is used.

Starting RPC from the inetd Daemon

An RPC server can be started from the **inetd** daemon. The only difference between using **inetd** and the usual code is that the service creation routine is called. Since the **inet** passes a socket as file descriptor 0 (zero), the following form is used:

```
transp = svcudp_create(0);          /*  For UDP                  */
transp = svctcp_create(0,0,0);      /*  For listener TCP sockets  */
transp = svcfd_create(0,0,0);       /*  For connected TCP sockets */
```

In addition, the **svc_register** routine should be called as follows:

```
svc_register(transp, PROGNUM, VERSNUM, service, 0)
```

The final flag is 0 (zero) since the program is already registered by the **inetd** daemon. To exit from the server process and return control to the **inet**, the user must explicitly exit. The **svc_run** routine never returns.

Entries in the **/etc/inetd.conf** file for RPC services take one of the following two forms:

```
p_name sunrpc_udp udp wait user server args version
```

```
p_name sunrpc_tcp tcp wait user server args version
```

where `p_name` is the symbolic name of the program as it appears in the RPC routine, `server` is the program implementing the server, and `version` is the version number of the service.

If the same program handles multiple versions, then the version number can be a range, as in the following:

```
rstatd sunrpc_udp udp wait root /usr/etc/rpc.rstatd rstatd 100001
1–2
```

Compiling and Linking RPC Programs

RPC subroutines are part of the **libc.a** library. Add the following line to the **Makefile** file:

```
CFLAGS=–D_BSD –DBSD_INCLUDES
```

Related Information

The **mount** command, **rpcgen** command, **spray** command.

The **inetd** daemon.

The **/etc/inetd.conf** file, **/etc/rpc** file.

The **svc_freeargs** macro, **svc_getargs** macro.

The **callrpc** subroutine, **clnttcp_create** subroutine, **clntudp_create** subroutine, **malloc** subroutine, **registerrpc** subroutine, **svc_register** subroutine, **svc_run** subroutine, **svctcp_create** subroutine, **svcudp_create** subroutine, **svc_run** subroutine, **xdr_array** subroutine, **xdr_string** subroutine.

List of RPC Examples on page 7–49.

Example Using Multiple Program Versions on page 7–73, Example Using the Highest Layer of RPC on page 7–58, Example Using the Intermediate Layer of RPC on page 7–59, Example Showing How RPC Passes Arbitrary Data Types on page 7–61, Example Using the Lowest Layer of RPC on page 7–63.

Understanding the rpcgen Protocol Compiler on page 7–37, Understanding the RPC Message Protocol on page 7–5.

eXternal Data Representation (XDR) Overview for Programming on page 3–1, Alphabetical List of XDR Subroutines and Macros on page 3–24, Functional List of XDR Subroutines and Macros on page 3–26.

Sockets Overview on page 9–1.

Understanding Protocols for TCP/IP, User Datagram Protocol in *Communication Concepts and Procedures*.

Understanding the RPC Features

The features of RPC include batching calls, broadcasting calls, call-back procedures, and using the **select** subroutine. Batching allows a client to send an arbitrarily large sequence of call messages to a server. Broadcasting allows a client to send a data packet to the network and wait for numerous replies. Call-back procedures permit a server to become a client and make an RPC call-back to the client's process. The **select** subroutine examines the I/O descriptor sets whose addresses are passed in the *readfds*, *writefds*, and *exceptfds* parameters to see if some of their descriptors are ready for reading or writing, or have an exceptional condition pending. It then returns the total number of ready descriptors in all the sets.

RPC is also used for the **rcp** program on TCP. See the Example Using rcp on TCP on page 7–69.

Batching Remote Procedure Calls

Batching allows a client to send an arbitrarily large sequence of call messages to a server. Batching typically uses reliable byte stream protocols, such as TCP/IP, for its transport. When batching, the client never waits for a reply from the server, and the server does not send replies to batched requests. Normally, a sequence of batch calls should be terminated by a legitimate, nonbatched RPC to flush the pipeline.

The RPC architecture is designed so that clients send a call message and then wait for servers to reply that the call succeeded. This implies that clients do not compute while servers are processing a call. However, the client may not want or need an acknowledgment for every message sent. Therefore, clients can use RPC batch facilities to continue computing while they wait for a response.

Batching can be thought of as placing RPC messages in a pipeline of calls to a desired server. Batching assumes the following:

- Each remote procedure call in the pipeline requires no response from the server, and the server does not send a response message.
- The pipeline of calls is transported on a reliable byte stream transport such as TCP/IP.

In order for a client to use batching, the client must perform remote procedure calls on a TCP-based transport. Batched calls must have the following attributes:

- The resulting XDR routine must be 0 (NULL).
- The remote procedure call's time out must be 0 (zero).

Since the server sends no message, the clients are not notified of any failures that occur. Therefore, clients must handle their own errors.

Since the server does not respond to every call, the client can generate new calls that run parallel to the server's execution of previous calls. Furthermore, the TCP/IP implementation can buffer many call messages, and send them to the server with one **write** system call. This overlapped execution decreases the interprocess communication overhead of the client and server processes as well as the total elapsed time of a series of calls. Batched calls are buffered, so the client should eventually perform a nonbatched remote procedure call in order to flush the pipeline with positive acknowledgment.

Broadcasting Remote Procedure Calls

In broadcast RPC-based protocols, the client sends a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses only packet-based protocols, such as UDP/IP, for its transports. Servers that support broadcast protocols respond only when the request is successfully processed and remain silent when errors occur. Broadcast RPC requires the RPC port mapper service to achieve its semantics. The **portmapper** daemon converts RPC program numbers into DARPA protocol port numbers. See the Example of Broadcasting a Remote Procedure Call on page 7–57.

The main differences between broadcast RPC and normal RPC are as follows:

- Normal RPC expects only one answer, while broadcast RPC expects one or more answers from each responding machine.
- The implementation of broadcast RPC treats unsuccessful responses as garbage by filtering them out. Therefore, if there is a version mismatch between the broadcaster and a remote service, the user of broadcast RPC may never know.
- All broadcast messages are sent to the port-mapping port. As a result, only services that register themselves with their port mapper are accessible through the broadcast RPC mechanism.
- Broadcast requests are limited in size to the maximum transfer unit (MTU) of the local network. For the Ethernet system, the MTU is 1500 bytes.
- Broadcast RPC is supported only by packet-oriented (connectionless) transport protocols such as UPD/IP.

Understanding RPC Call-back Procedures

Occasionally, the server may need to become a client by making an RPC call-back to the client's process. In order to do an RPC call-back, the user needs a program number on which to make the remote procedure call. Since this is a dynamically generated program number, it should be in the transient range, 0x40000000 to 0x5fffffff. See the Example Using Call-back Procedures on page 7–84 for more information.

Understanding the select Subroutine on the Server Side

The **select** subroutine checks the specified file descriptors and message queues to see if they are ready for reading (receiving) or writing (sending), or if they have an exceptional condition pending. A **select** procedure allows the server to interrupt an activity, check for data, and then continue processing the activity. For example, if the server processes RPC requests while performing some other activity that involves periodically updating a data structure, the process can set an alarm signal to notify the server before calling the **svc_run** routine. However, if the current activity is waiting on a file descriptor, the call to the **svc_run** routine does not work. See the Example Using the select Subroutine on page 7–68 for more information.

The programmer can bypass the **svc_run** routine and call the **svc_getreqset** routine directly. It is necessary to know the file descriptors of the socket(s) associated with the programs being waited on. The programmer can have a **select** statement that waits on both the RPC socket and his own descriptors. Note that the **svc_fds** parameter is a bit mask of all the file descriptors that RPC is using for services. It can change each time that any RPC library routine is called, because descriptors are continually opened and closed. TCP connections are an example.

Related Information

The **portmap** daemon.

The **gettransient** subroutine, **pmap_set** subroutine, **select** subroutine, **svctcp_create** subroutine, **svcudp_create** subroutine, **svc_run** subroutine.

The **write** system call.

List of RPC Examples on page 7–49.

Example Using RPC Callback Procedures on page 7–84, Example Using the select Subroutine on page 7–68, Example of Broadcasting a Remote Procedure Call on page 7–57, Example Using rcp on TCP on page 7–69.

Understanding the Port Mapper Program on page 7–17, Using the RPC Message Protocol on page 7–5.

Understanding Protocols for TCP/IP in *Communication Concepts and Procedures.*

Understanding the RPC Language

The Remote Procedure Call Language (RPCL) is identical to the XDR language, except for the added program definition.

Understanding RPC Language Descriptions

Just as the eXternal Data Representation (XDR) data types are described in a formal language, it is necessary to describe the procedures that operate on these XDR data types in a formal language. As an extension to the XDR language, the RPCL is used for this purpose.

RPC uses the RPCL as the input language to its protocol and routines. RPCL specifies the data types used by RPC and generates the XDR routines that standardize their representation. In order to implement the service protocols and routines, the RPCL input is compiled into the corresponding C language code, using the **rpcgen** command.

The RPC language descriptions include:

- Definitions
- Structures
- Unions
- Enumerations
- Typedefs
- Constants
- Programs
- Declarations.

See the Example of an RPC Language ping Program on page 7–56 for more information. In addition to the language descriptions, there are some exceptions to the rules for the RPC Language.

Definitions

An RPC language file consists of a series of definitions in the following format:

```
definition-list:
     definition ";"
     definition ";" definition-list
```

RPC recognizes the following six types of definitions:

```
definition:
      enum-definition
      struct-definition
      union-definition
      typedef-definition
      const-definition
      program-definition
```

Structures

The C language structures are usually located in header files located in the **/usr/include** or **/usr/include/sys** directories, although the structures can be located in any directory in the file system. An XDR structure, declared almost exactly like its C counterpart, appears as the following:

```
struct-definition:
      "struct" struct-ident "{"
      declaration-list
      "}"

declaration-list:
      declaration ";"
      declaration ";" declaration-list
```

As an example, here is an XDR structure to a two-dimensional coordinate, and the C structure that it gets compiled into in the output header file:

```
struct coord {           struct coord {
      int x;     -->           int x;
      int y;                   int y;

};                       };
                         typedef struct coord coord;
```

The output is identical to the input, except for the added `typedef` at the end of the output. This allows the programmer to use `coord` instead of `struct coord` when declaring items.

Unions

XDR unions are discriminated unions and look quite different from C unions. They are more analogous to Pascal variant records than to C unions. The following is a union definition:

```
union-definition:
      "union" union-ident "switch" "(" declaration ")" "{"
           case-list
      "}"

case-list:
      "case" value ":" declaration ";"
      "default" ":" declaration ";"
      "case" value ":" declaration ";" case-list
```

The following is an example of a type that might be returned as the result of a `read data` operation. If there is no error, it returns a block of data; otherwise, it returns nothing.

```
union read_result switch (int errno) {
case 0
     opaque data[1024];
default:
     void;
};
```

It gets compiled into the following:

```
struct read_result {
     int errno;
     union {
          char data[1024];
     }read_result_u;
};
typedef struct read_result read_result;
```

Notice that the union component of the output structure has the same name as the type name, except for the trailing `_u`.

Enumerations

XDR enumerations have the same syntax as C enumerations:

```
enum–definition:
      "enum" enum–ident "{"
      enum–value–list
      "}"

enum–value–list:
      enum–value
      enum–value "," enum–value–list

enum–value:
      enum–value–ident
      enum–value–ident "=" value
```

The following is a short example of an XDR enumeration, and the C enumeration that it gets compiled into:

```
enum colortype {            enum colortype {
     RED = 0,                    RED = 0,
     GREEN = 1,      —>          GREEN = 1,
     BLUE = 2                    BLUE = 2,
};                          };
                            typedef enum colortype colortype;
```

Type Definitions

XDR type definitions (typedefs) have the same syntax as C typedefs:

```
typedef–definition:
      "typedef" declaration
```

The following is an example that defines an `fname_type` used for declaring file name strings that have a maximum length of 255 characters:

```
typedef string fname_type<255>; —> typedef char *fname_type;
```

Constants

XDR constants can be used wherever an integer constant is required. The definition for a constant is:

```
const-definition:
      "const" const-ident "=" integer
```

For example, the following defines a constant `DOZEN` equal to 12:

```
const DOZEN = 12; --> #define DOZEN 12
```

Programs

RPC programs are declared using the following syntax:

```
program-definition:
      "program" program-ident "{"
           version-list
      "}" "=" value

version-list:
      version ";"
      version ";" version-list

version:
      "version" version-ident "{"
           procedure-list
      "}" "=" value

procedure-list:
      procedure ";"
      procedure ";" procedure-list

procedure:
      type-ident procedure-ident "(" type-ident ")" "=" value
```

The the time protocol is defined as follows:

```
/*
 * time.x: Get or set the time. Time is represented as number
 * of seconds since 0:00, January 1, 1970.
 */
program TIMEPROG {
     version TIMEVERS {
          unsigned int TIMEGET (void) = 1;
          void TIMESET (unsigned) = 2;
     } = 1;
} = 44;
```

This file compiles into `#defines` in the output header file:

```
#define TIMEPROG 44

#define TIMEVERS 1

#define TIMEGET 1

#define TIMESET 2
```

Declarations

In XDR, there are four types of declarations: simple declarations, fixed array declarations, variable array declarations, and pointer declarations. Declarations have the following forms:

```
declaration:
      simple-declaration
      fixed-array-declaration
      variable-array-declaration
      pointer-declaration
```

Simple Declarations

Simple XDR declarations are like simple C declarations, as follows:

```
simple-declaration:
      type-ident variable-ident
```

Example:

```
colortype color;  -->  colortype color;
```

Fixed-Length Array Declarations

Fixed-length array declarations are just like C array declarations, as follows:

```
fixed-array-declaration:
      type-ident variable-ident "[" value "]"
```

Example:

```
colortype palette[8];  -->  colortype palette[8]
```

Variable-Length Array Declarations

Variable-length array declarations have no explicit syntax in C, so XDR invents its own syntax using angle brackets. The maximum size is specified between the angle brackets. A specific size can be omitted to indicate that the array may be of any size.

```
variable-array-declaration:
      type-ident variable-ident "<" value ">"
      type-ident variable-ident "<" ">"
```

The maximum size is specified between the angle brackets. The size may be omitted, indicating that the array may be of any size.

```
      int heights<12>;          /* at most 12 items     */
      int widths<>;             /* any number of items  */
```

Since variable-length arrays have no explicit syntax in C, these declarations are actually compiled into structure definitions (`struct`). For example, the `heights` declaration gets compiled into the following structure:

```
struct {
     u_int heights_len;    /* # of items in array */
     int *heights_val;     /* # pointer to array */
} heights;
```

Pointer Declarations

Pointer declarations are made in XDR exactly as they are in C. The programmer cannot send pointers over a network, but can use XDR pointers for sending recursive data types such as lists and trees. In XDR language, the type is called `optional-data`, instead of `pointer`. Pointer declarations have the following form in XDR language:

```
pointer-declaration:
      type-ident "*" variable-ident
```

Example:

```
listitem *next;  -->  listitem *next;
```

RPCL Syntax Requirements for Program Definition

The RPCL has the following syntax requirements:

- The `program` and `version` keywords are added and cannot be used as identifiers.
- A version name cannot occur more than once within the scope of a program definition. Nor can a version number occur more than once within the scope of a program definition.
- A procedure name cannot occur more than once within the scope of a version definition. Nor can a procedure number occur more than once within the scope of a version definition.
- Program identifiers are in the same name space as the `constant` and `type` identifiers.
- Only unsigned constants can be assigned to `program`, `version`, and `procedure` definitions.

Exceptions to the RPCL Rules

The exceptions to the rules of the RPC language include Booleans, strings, opaque data, and voids:

Booleans

The C language has no built-in Boolean type. However, the RPC library uses a Boolean type called `bool_t`, which is either TRUE or FALSE. Objects that are declared as type `bool` in XDR language are compiled into `bool_t` in the output header file.

Example:

```
bool married;  -->  bool_t married;
```

Strings

The C language has no built-in string type. Instead, it uses the null-terminated `char *` convention. In the XDR language, strings are declared using the `string` keyword, and then compiled into `char *` in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (not counting the NULL character). The maximum size may be left off, indicating a string of arbitrary length.

Examples:

```
string name<32>;  -->  char *name;

string longname<>;  -->  char *longname;
```

Opaque Data

Opaque data is used in RPC and XDR to describe untyped data: that is, sequences of arbitrary bytes. Opaque data may be declared either as a fixed-length or variable-length array.

Examples:

```
opaque diskblock[512];    —>  char diskblock[512];

opaque filedata<1024>;    —>  struct {
                                  u_int filedata_len;
                                  char *filedata_val;
                              } filedata
```

Voids

In a void declaration, the variable is not named. The declaration is simply `void`. Void declarations can occur as the argument or result of a remote procedure in only two places: union definitions and program definitions.

Related Information

The **rpcgen** command.

List of RPC Examples on page 7–49.

Example of an RPC Language ping Program on page 7–56.

Understanding the RPC Message Protocol on page 7–5, Understanding the rpcgen Protocol Compiler on page 7–37.

eXternal Data Representation (XDR) Overview for Programming on page 3–1, Functional List of XDR Subroutines and Macros on page 3–26.

Understanding the rpcgen Protocol Compiler

The **rpcgen** protocol compiler accepts a remote program interface definition written in the RPC language, which is similar to the C language. The **rpcgen** compiler helps programmers write RPC applications simply and directly. The **rpcgen** compiler debugs the network interface code, thereby allowing programmers to spend their time debugging the main features of their applications.

The **rpcgen** compiler produces a C language output that includes the following:

- Stub versions of the client and server routines
- Server skeleton
- XDR filter routines for parameters and results
- A header file that contains common definitions of constants and macros.

Client stubs interface with the RPC library to effectively hide the network from its callers. Server stubs similarly hide the network from the server procedures that are to be invoked by remote clients. The **rpcgen** output files can be compiled and linked in the usual way. The programmer writes server procedures in any language and then links the procedures with the server skeleton to get an executable server program.

When application programs use the **rpcgen** compiler, there are many details to consider. Of particular importance is the writing of XDR routines needed to convert procedure arguments and results into the network format, and vice versa.

This discussion of the **rpcgen** protocol compiler includes the following topics:

- Converting Local Procedures into Remote Procedures
- Generating XDR Routines
- Understanding the C Preprocessor.
- Changing Time Outs
- Handling Broadcast on the Server Side
- Other Information Passed to Server Procedures

Converting Local Procedures into Remote Procedures

Applications running at a single workstation can be converted to run over the network. A converted procedure can be called from anywhere in the network. Generally, it is necessary to identify the types for all procedure inputs and outputs. A null procedure (procedure 0) is not necessary because the **rpcgen** compiler generates it automatically. See the Example Converting Local Procedures into Remote Procedures on page 7–75 for more information.

Generating XDR Routines

The **rpcgen** compiler can be used to generate XDR routines that are necessary to convert local data structures into network format, and vice versa. Some types can be defined using the `struct`, `union`, and `enum` keywords. However, these keywords should not be used in subsequent declarations of variables of these same types. The **rpcgen** compiler compiles RPC unions into C structures. It is an error to declare these unions using the `union` keyword. See the Example Generating an XDR Routine on page 7–80 for more information.

Understanding the C Preprocessor

The C language preprocessor is run on all input files before they are compiled, so all the preprocessor directives are legal within a **.x** file. Four symbols can be defined, depending upon which output file is generated. The symbols and their uses are:

RPC_HDR Represents header file output.

RPC_XDR Represents XDR routine output.

RPC_SVC Represents server skeleton output.

RPC_CLNT Represents client stub output.

The **rpcgen** compiler also does some preprocessing of its own. Any line that begins with a % (percent sign) is passed directly into the output file without an interpretation of the line. Use of the percent feature is not generally recommended, since there is no guarantee that the compiler will put the output where it is intended.

Changing Time Outs

When using the **clnt_create** routine, RPC sets a default time out of 25 seconds for remote procedure calls. The time-out default can be changed using the **clnt_control** routine. The following code fragment illustrates the use of this routine:

```
struct timeval tv
CLIENT *cl;
cl=clnt_create("somehost", SOMEPROG, SOMEVERS, "tcp");
if (cl=NULL) {
  exit(1);
}
tv.tv_sec=60; /* change timeout to 1 minute */
tv.tv_usec=0;
clnt_control(cl, CLSET_TIMEOUT, &tv);
```

Handling Broadcast on the Server Side

When a client calls a procedure through broadcast RPC, the server normally replies only if it can provide useful information to the client. This prevents flooding the network with useless replies.

To prevent the server from replying, a remote procedure can return NULL as its result. The server code generated by the **rpcgen** compiler detects this and does not send a reply. For example, the following procedure replies only if it thinks it is a server:

```
void *
reply_if_nfsserver()
{
   char notnull; /* just here so we can use its address */
   if {access("/etc/exports", F_OK) < 0) {
      return (NULL); /* prevent RPC from replying */
   }
   /*
   *return non-null pointer so RPC will send out a reply
   */
   return ((void *) &notnull);
}
```

If a procedure returns type `void`, the server must return a non-null pointer in order for RPC to reply.

Other Information Passed to Server Procedures

Server procedures often want more information about a remote procedure call than just its arguments. For example, getting authentication information is important to procedures that implement some level of security. This additional information is supplied to the server procedure as a second argument.

The following example program that allows only root users to print a message on the console, demonstrates its use.

```
int *
printmessage_1(msg, rq)
   char **msg;
   struct svc_req *rq;
{
   static in result;  /* Must be static */
   FILE *f;
   struct authunix_parms *aup;
   aup=(struct authunix_parms *)rq->rq_clntcred;
   if (aup->aup_uid !=0) {
      result=0;
      return (&result);
   }
   /*
   *Same code as before.
   */
}
```

Related Information

The **rpcgen** command.

The **clnt_create** subroutine.

List of RPC Examples on page 7–49.

Example Generating an XDR Routine on page 7–80, Example Converting Local Procedures into Remote Procedures on page 7–75.

Understanding the RPC Language on page 7–31, Understanding the RPC Message Protocol on page 7–5.

eXternal Data Representation (XDR) Overview for Programming on page 3–1, Functional List of XDR Subroutines and Macros on page 3–26.

Alphabetical List of RPC Subroutines and Macros

auth_destroy	Destroys authentication information.
authdes_create	Enables the use of DES from the client side.
authdes_getucred	Maps a DES credential into a UNIX credential.
authnone_create	Creates NULL authentication information.
authunix_create	Creates an authentication handle with AIX permissions.
authunix_create_default	Sets the authentication to the default.
callrpc	Calls the remote procedure on the machine associated with *host*.
clnt_broadcast	Broadcasts a remote procedure call to all network hosts.
clnt_call	Calls the remote procedure associated with *clnt*.
clnt_control	Changes or retrieves information about a client object.
clnt_create	Creates a generic client transport handle.
clnt_destroy	Destroys a client's RPC handle.
clnt_freeres	Frees memory allocated by RPC and XDR.
clnt_geterr	Copies error information from a client transport handle.
clnt_pcreateerror	Identifies why a client RPC handle was not created.
clnt_perrno	Specifies the condition of the *stat* parameter.
clnt_perror	Determines why a remote procedure call failed.
clnt_spcreateerror	Identifies why a client RPC handle was not created.
clnt_sperrno	Specifies the condition of the *stat* parameter.
clnt_sperror	Indicates why a remote procedure failed.
clntraw_create	Creates a sample RPC client handle for simulation.
clnttcp_create	Creates a TCP/IP client transport handle.
clntudp_create	Creates a UDP/IP client transport handle.
get_myaddress	Gets the user's IP address.
getnetname	Installs the network name of the caller in the array.
host2netname	Converts a host name to a network name.
key_decryptsession	Decrypts a server network name and DES key.

key_encryptsession	Encrypts a server network name and DES key.
key_gendes	Requests a secure conversation key from the **keyserv** daemon.
key_setsecret	Sets the key for the user ID of the calling process.
netname2host	Converts a network name to a host name.
netname2user	Converts a network name to a user ID.
pmap_getmaps	Returns a list of the current RPC port mappings.
pmap_getport	Requests the port number on which a service waits.
pmap_rmtcall	Instructs the **portmap** daemon to make an RPC.
pmap_set	Maps a remote procedure call to a port.
pmap_unset	Destroys the mapping between the RPC and the port.
registerrpc	Registers a procedure with the RPC service.
rtime	Returns the remote time in the **timeval** structure.
svc_destroy	Destroys a service transport handle.
svc_freeargs	Frees data allocated by the RPC and XDR system.
svc_getargs	Decodes the arguments of an RPC request.
svc_getcaller	Gets the network address of the caller of a procedure.
svc_getreqset	Services an RPC request.
svc_register	Maps a remote procedure.
svc_run	Signals a wait for the arrival of RPC requests.
svc_sendreply	Sends back the results of a remote procedure call.
svc_unregister	Removes mappings between procedures and objects.
svcerr_auth	Indicates that the remote procedure call cannot be completed due to an authentication error.
svcerr_decode	Indicates that the parameters of a request cannot be decoded.
svcerr_noproc	Indicates that the remote procedure call cannot be completed because the program cannot support the requested procedure.
svcerr_noprog	Indicates that the remote procedure call cannot be completed because the program is not registered.
svcerr_progvers	Indicates that the remote procedure call cannot be completed because the program version is not registered.

svcerr_systemerr Indicates that the remote procedure call cannot be completed due to an error not covered by any protocol.

svcerr_weakauth Indicates that the remote procedure call cannot be completed due to insufficient authentication security parameters.

svcfd_create Creates a service on any open file descriptor.

svcraw_create Creates a sample RPC service handle for simulation.

svctcp_create Creates a TCP/IP service transport handle.

svcudp_create Creates a UDP/IP service transport handle.

user2netname Converts a user ID to a network name.

xdr_accepted_reply Encodes an RPC reply messages.

xdr_authunix_parms Describes UNIX-style credentials.

xdr_callhdr Describes RPC call header messages.

xdr_callmsg Describes RPC call messages.

xdr_opaque_auth Describes RPC authentication messages.

xdr_pmap Describes parameters for **portmap** procedures.

xdr_pmaplist Describes a list of port mappings externally.

xdr_rejected_reply Describes RPC rejected message replies.

xdr_replymsg Describes RPC message replies.

xprt_register Registers an RPC service transport handle.

xprt_unregister Removes an RPC service transport handle.

Related Information

Functional List of RPC Subroutines and Macros on page 7–44.

List of RPC Examples on page 7–49.

Programming in RPC on page 7–20.

Functional List of RPC Subroutines and Macros

RPC provides subroutines and macros for performing various tasks. The following list is a functional grouping of the RPC subroutines and macros:

- Authenticating Remote Procedure Calls
- Managing the Client
- Managing the Server
- Using RPC Utilities
- Using DES Interface to the keyserv Daemon
- Interfacing to the portmap Daemon
- Describing and Encoding Remote Procedure Calls.

Authenticating Remote Procedure Calls

RPC provides subroutines and macros for creating and destroying authentication information.

Creating RPC Authentication Information

The following RPC routines create authentication information:

authnone_create	Creates NULL authentication information.
authunix_create	Creates an authentication handle with AIX permissions.
authunix_create_default	Sets the authentication to the default.
authdes_create	Enables the use of DES from the client side.
authdes_getucred	Maps a DES credential into a UNIX credential.

Destroying RPC Authentication Information

The following RPC routine destroys authentication information:

auth_destroy	Destroys authentication information.

Managing the Client

RPC provides subroutines and macros for the following client management tasks:

- Creating an RPC client for a remote program
- Changing or retrieving client information
- Destroying a client RPC handle
- Broadcasting a remote procedure call
- Calling a remote procedure
- Freeing memory allocated by RPC and XDR
- Handling client errors.

Creating an RPC Client for a Remote Program

The following RPC routines create client transport handles:

clntraw_create Creates a sample RPC client handle for simulation.

clnttcp_create Creates a TCP/IP client transport handle.

clntudp_create Creates a UDP/IP client transport handle.

clnt_create Creates a generic client transport handle.

Changing or Retrieving Client Information

The following RPC routine changes or retrieves client information:

clnt_control Changes or retrieves information about a client object.

Destroying a Client RPC Handle

The following RPC routine destroys a client transport handle:

clnt_destroy Destroys a client's RPC handle.

Broadcasting a Remote Procedure Call

The following RPC routine broadcasts calls:

clnt_broadcast Broadcasts a remote procedure call to all network hosts.

Calling a Remote Procedure

The following RPC routines call a remote procedure:

callrpc Calls the remote procedure on the machine associated with the *host* parameter.

clnt_call Calls the remote procedure associated with the *clnt* parameter.

Freeing Memory Allocated by RPC and XDR

The following RPC routine frees allocated memory:

clnt_freeres Frees memory allocated by RPC and XDR.

Handling Client Errors

The following RPC routines handle errors on the client:

clnt_pcreateerror Identifies why a client RPC handle was not created.

clnt_perrno Specifies the condition of the *stat* parameter.

clnt_perror Determines why a remote procedure call failed.

clnt_geterr Copies error information from a client transport handle.

clnt_spcreateerror Identifies why a client RPC handle was not created.

clnt_sperrno Specifies the condition of the *stat* parameter.

clnt_sperror Indicates why a remote procedure call failed.

Managing the Server

RPC provides subroutines and macros for the following server management tasks:

- Creating an RPC service transport handle
- Destroying an RPC service transport handle
- Registering and unregistering RPC procedures and handles
- Handling an RPC request
- Handling server errors.

Creating an RPC Service Transport Handle

The following RPC routines create service transport handles:

svcraw_create	Creates a sample RPC service handle for simulation.
svctcp_create	Creates a TCP/IP service transport handle.
svcudp_create	Creates a UDP/IP service transport handle.
svcfd_create	Creates a service on any open file descriptor.

Destroying an RPC Service Transport Handle

The following RPC routine destroys a service handle:

svc_destroy	Destroys a service transport handle.

Registering and Unregistering RPC Procedures and Handles

The following RPC routines register and map procedures and handles:

registerrpc	Registers a procedure with the RPC service.
xprt_register	Registers an RPC service transport handle.
xprt_unregister	Removes an RPC service transport handle.
svc_register	Maps a remote procedure.
svc_unregister	Removes mappings between procedures and objects.

Handling an RPC Request

The following routines handle RPC requests:

svc_run	Signals a wait for the arrival of RPC requests.
svc_getreqset	Services an RPC request.
svc_getargs	Decodes the arguments of an RPC request.
svc_sendreply	Sends back the results of a remote procedure call.
svc_freeargs	Frees data allocated by the RPC and XDR system.
svc_getcaller	Gets the network address of the caller of a procedure.

Handling Server Errors

The following RPC routines handle errors on the server:

svcerr_auth	Indicates that the remote procedure call cannot be completed due to an authentication error.
svcerr_decode	Indicates that the parameters of a request cannot be decoded.
svcerr_noproc	Indicates that the remote procedure call cannot be completed because the program cannot support the requested procedure.
svcerr_noprog	Indicates that the remote procedure call cannot be completed because the program is not registered.
svcerr_progvers	Indicates that the remote procedure call cannot be completed because the program version is not registered.
svcerr_systemerr	Indicates that the remote procedure call cannot be completed due to an error not covered by any protocol.
svcerr_weakauth	Indicates that the remote procedure call cannot be completed due to insufficient authentication security parameters.

Using RPC Utilities

RPC provides the following RPC utilities:

host2netname	Converts a host name to a network name.
netname2host	Converts a network name to a host name.
netname2user	Converts a network name to a user ID.
user2netname	Converts a user ID to a network name.
getnetname	Installs the network name of the caller in the array.
get_myaddress	Gets the user's IP address.
rtime	Returns the remote time in the **timeval** structure.

Using DES Interface to the keyserv Daemon

RPC provides subroutines for interfacing to the **keyserv** daemon:

key_decryptsession	Decrypts a server network name and a DES key.
key_encryptsession	Encrypts a server network name and a DES key.
key_gendes	Requests a secure conversation key from the **keyserv** daemon.
key_setsecret	Sets the key for the user ID of the calling process.

Interfacing to the portmap Daemon

RPC provides subroutines for interfacing to the **portmap** daemon:

pmap_getmaps	Returns a list of the current RPC port mappings.
pmap_getport	Requests the port number on which a service waits.
pmap_rmtcall	Instructs the **portmap** daemon to make an RPC.
pmap_set	Maps a remote procedure call to a port.
pmap_unset	Destroys the mapping between the RPC and the port.
xdr_pmap	Describes parameters for **portmap** procedures.
xdr_pmaplist	Describes a list of port mappings externally.

Describing and Encoding Remote Procedure Calls

RPC provides subroutines for describing and encoding RPC call and reply messages, authentication, and portmappings:

xdr_accepted_reply	Encodes RPC reply messages.
xdr_authunix_parms	Describes UNIX-style credentials.
xdr_callhdr	Describes RPC call header messages.
xdr_callmsg	Describes RPC call messages.
xdr_opaque_auth	Describes RPC authentication messages.
xdr_rejected_reply	Describes RPC message rejection replies.
xdr_replymsg	Describes RPC message replies.

Related Information

Alphabetical List of RPC Subroutines and Macros on page 7–41.

List of RPC Examples on page 7–49.

Programming in RPC on page 7–20.

Remote Procedure Call (RPC) Overview for Programming on page 7–1.

List of RPC Examples

Example Converting Local Procedures into Remote Procedures on page 7–75

Example Generating XDR Routines on page 7–80

Example of an RPC Language ping Program on page 7–56

Example of Broadcasting a Remote Procedure Call on page 7–57

Example Showing How RPC Passes Arbitrary Data on page 7–61

Example Using DES Authentication on page 7–53

Example Using Multiple Program Versions on page 7–73

Example Using rcp on TCP on page 7–69

Example Using RPC Callback Procedures on page 7–84

Example Using the Highest Layer of RPC on page 7–58

Example Using the Intermediate Layer of RPC on page 7–59

Example Using the Lowest Layer of RPC on page 7–63

Example Using the select Subroutine on page 7–68

Example Using UNIX Authentication on page 7–50

Related Information

Alphabetical List of RPC Subroutines and Macros on page 7–41, Functional List of RPC Subroutines and Macros on page 7–44.

Example Using UNIX Authentication

This example shows how UNIX authentication works on both the client and server sides.

UNIX Authentication on the Client Side

To use UNIX authentication, the programmer first creates the RPC client handle and then set the authentication parameter. The RPC client handle can be created as follows:

```
clnt = clntudp_create (address, prognum, versnum, wait, sockp)
```

The UNIX authentication parameter can be set as follows:

```
clnt->cl_auth = authunix_create_default();
```

Each remote procedure call associated with the client (`clnt`) then carries the following UNIX-style authentication credentials structure:

```
/*
 * UNIX style credentials.
 */
struct authunix_parms {
    u_long  aup_time;        /*  credentials creation time       */
    char    *aup_machname;   /*  host name where client is       */
    int     aup_uid;         /*  client's UNIX effective uid     */
    int     aup_gid;         /*  client's current group id       */
    u_int   aup_len;         /*  element length of aup_gids      */
    int     *aup_gids;       /*  array of groups user is in      */
};
```

The **authunix_create_default** subroutine sets these fields by invoking the appropriate subroutines. The UNIX-style authentication is valid until it is destroyed with the following routine:

```
auth_destroy(clnt->cl_auth);
```

UNIX Authentication on the Server Side

The following example shows how to use UNIX authorization on the server side.

The following is a structure definition of a request handle passed to a service dispatch routine at the server:

```
/*
 * An RPC Service request
 */
struct svc_req {
    u_long    rq_prog;          /*  service program number        */
    u_long    rq_vers;          /*  service protocol vers num     */
    u_long    rq_proc;          /*  desired procedure number      */
   struct opaque_auth rq_cred;  /*  raw credentials from wire     */
   caddr_t   rq_clntcred;       /*  credentials (read only)       */
};
```

Except for the style or flavor of authentication credentials, the rq_cred routine is opaque.

```
/*
 * Authentication info. Mostly opaque to the programmer.
 */
struct opaque_auth {
    enum_t  oa_flavor;  /*  style of credentials                */
    caddr_t oa_base;    /*  address of more auth stuff          */
    u_int   oa_length;  /*  not to exceed MAX_AUTH_BYTES        */
};
```

Before passing a request to the service dispatch routine, RPC guarantees:

- The request's rq_cred field is in an acceptable form. Therefore, the service implementor may inspect the request's rq_cred.oa_flavor to determine which style of authentication the caller used. The service implementor may also wish to inspect the other rq_cred fields if the authentication style is not one of the styles supported by the RPC package.
- The request's rq_clntcred field is either NULL or points to a well formed structure that corresponds to a supported style of authentication credentials. The rq_clntcred field can currently be set as a pointer to an authunix_parms structure for UNIX style authentication. If rq_clntcred is NULL, the service implementor can inspect the other opaque fields of the rq_cred credential for any new types of authentication that may be unknown to the RPC package.

The following is an example using UNIX authentication on the server side.The remote users service example can be extended, so that it computes results for all users except UID 16:

```
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    struct authunix_parms *unix_cred;
    int uid;
    unsigned long nusers;

    /*
     * we don't care about authentication for null proc
     */
    if (rqstp->rq_proc == NULLPROC) {
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can't reply to RPC call\n");
            return (1);
        }
        return;
    }
```

```
        /*
         * now get the uid
         */
        switch (rqstp->rq_cred.oa_flavor) {
        case AUTH_UNIX:
            unix_cred =
                (struct authunix_parms *)rqstp->rq_clntcred;
            uid = unix_cred->aup_uid;
            break;
        case AUTH_NULL:
        default:
            svcerr_weakauth(transp);
            return;
        }
        switch (rqstp->rq_proc) {
        case RUSERSPROC_NUM:
            /*
             * make sure caller is allowed to call this proc
             */
            if (uid == 16) {
                svcerr_systemerr(transp);
                return;
            }
            /*
             * Code here to compute the number of users
             * and assign it to the variable nusers
             */
            if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
                fprintf(stderr, "can't reply to RPC call\n");
                return (1);
            }
            return;
        default:
            svcerr_noproc(transp);
            return;
        }
    }
```

Related Information

The **auth_destroy** macro.

The **authnone_create** subroutine, **authunix_create** subroutine, **clntudp_create** subroutine, **svcerr_noproc** subroutine, **svcerr_systemerr** subroutine, **svcerr_weakauth** subroutine, **svc_sendreply** subroutine, **xdr_u_long** subroutine.

List of RPC Examples on page 7-49.

Understanding UNIX Authentication on page 7-11.

Example Using DES Authentication

The following example illustrates how DES authentication works on both the client side and the server side.

DES Authentication on the Client Side

To use DES authentication, the client first sets its authentication handle as follows:

```
cl->cl_auth =
    authdes_create(servername, 60, &server_addr, NULL);
```

The first argument (servername) to the **authdes_create** routine is the network name or netname of the owner of the server process. Typically, server processes are root processes. The netname can be derived using the following call:

```
char servername[MAXNETNAMELEN];

host2netname(servername, rhostname, NULL);
```

The rhostname parameter is the host name of the machine on which the server process is running. The host2netname routine supplies the servername that contains this netname for the root process. If the server process is run by a regular user, the user2netname routine can be called instead.

The following example illustrates a server process with the same user ID as the client:

```
char servername[MAXNETNAMELEN];

user2netname(servername, getuid(), NULL);
```

The user2netname and host2netname routines identify the naming domain at the server location. The NULL parameter in this example means that the local domain name should be used.

The second argument (60) to the **authdes_create** routine identifies the lifetime of the credential, which is 60 seconds. This means the credential has 60 seconds until expiration. The server RPC subsystem does not grant either a second request within the 60-second lifetime or requests made after the credential has expired.

The third argument (&server_addr) to the **authdes_create** routine is the address of the host with which to synchronize. DES authentication requires that the server and client agree upon the time. The time is determined by the server when it receives the address. If the server and client times are already synchronized, the argument can be set to NULL.

The final argument (NULL) to the **authdes_create** routine is the address of a DES encryption key that is used to encrypt timestamps and data. If this argument is NULL, a random key is chosen. The programmer can get the encryption key from the *ah_key* field of the authentication handle.

DES Authentication on the Server Side

This example illustrates DES authentication on the server side. The server side is simpler than the client side. The following example uses AUTH_DES instead of AUTH_UNIX:

```
#include <sys/time.h>
#include <rpc/auth_des.h>
    ...
    ...
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    struct authdes_cred *des_cred;
    int uid;
    int gid;
    int gidlen;
    int gidlist[10];
    /*
     * we don't care about authentication for null proc
     */


    if (rqstp->rq_proc == NULLPROC) {
        /*  same as before  */
    }

    /*
     * now get the uid
     */
    switch (rqstp->rq_cred.oa_flavor) {
    case AUTH_DES:
        des_cred =
            (struct authdes_cred *) rqstp->rq_clntcred;
        if (! netname2user(des_cred->adc_fullname.name,
            &uid, &gid, &gidlen, gidlist))
        {
            fprintf(stderr, "unknown user: %s\n",
                des_cred->adc_fullname.name);
            svcerr_systemerr(transp);
            return;
        }
        break;
    case AUTH_NULL:
    default:
        svcerr_weakauth(transp);
        return;
    }
```

```
    /*
     * The rest is the same as UNIX-style authentication
     */
    switch (rqstp->rq_proc) {
    case RUSERSPROC_NUM:
        /*
         * make sure caller is allowed to call this proc
         */
        if (uid == 16) {
            svcerr_systemerr(transp);
            return;
        }
        /*
         * Code here to compute the number of users
         * and assign it to the variable nusers
         */
        if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
            fprintf(stderr, "can't reply to RPC call\n");
            return (1);
        }
        return;
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

Note the use of the `netname2user` routine, the inverse of the `user2netname` routine: it takes a network ID and converts to a user ID. The `netname2user` routine also supplies the group IDs, which are not used in this example, but which may be useful to other programs.

Related Information

The **authdes_create** subroutine, **host2netname** subroutine, **netname2user** subroutine, **svcerr_noproc** subroutine, **svcerr_systemerr** subroutine, **svcerr_weakauth** subroutine, **svc_sendreply** subroutine, **user2netname** subroutine, **xdr_u_long** subroutine.

List of RPC Examples on page 7–49.

Understanding Data Encryption Standard (DES) Authentication on page 7–12.

Example of an RPC Language ping Program

The following is an example of the specification of a simple ping program described in the RPC language.

```
/*
* Simple ping program
*/
program PING_PROG {
        /* Latest and greatest version */
        version PING_VERS_PINGBACK {
         void
        PINGPROC_NULL(void) = 0;

      /*
       * Ping the caller, return the round-trip time
       * (in microseconds).  Returns -1 if the operation
       * timed out.
       */
           int
           PINGPROC_PINGBACK(void) = 1;
} = 2;

/*
* Original version
*/
version PING_VERS_ORIG {
        void
        PINGPROC_NULL(void) = 0;
        } = 1;
} = 1;

const PING_VERS = 2;      /* latest version */
```

In this example, the first of the ping program (PING_VERS_PINGBACK) has two procedures: PINGPROC_NULL and PINGPROC_PINGBACK. The PINGPROC_NULL procedure takes no arguments and returns no results. However, it is useful for computing round-trip times from the client to the server. By convention, procedure 0 of an RPC protocol should have the same semantics and require no kind of authentication. The second procedure (PINGPROC_PINGBACK) requests a reverse ping operation from the server. It returns the amount of time in microseconds that the operation used.

The original version of the ping program, PING_VERS_ORIG, does not contain the PINGPROC_PINGBACK procedure. The original version is useful for compatibility with older client programs. As this program matures, it may be dropped from the protocol entirely.

Related Information

List of RPC Examples on page 7–49.

Understanding the RPC Language on page 7–31.

Example of Broadcasting a Remote Procedure Call

The following example illustrates broadcast RPC:

```
#include <rpc/pmap_clnt.h>
    ...
enum clnt_stat     clnt_stat;
    ...
clnt_stat = clnt_broadcast(prognum, versnum, procnum,
  inproc, in, outproc, out, eachresult)
    u_long    prognum;      /* program number             */
    u_long    versnum;      /*  version number            */
    u_long    procnum;      /*  procedure number          */
    xdrproc_t inpro         /*  xdr routine for args      */
    caddr_t   in;           /*  pointer to args           */
    xdrproc_t outproc       /*  xdr routine for results   */
    caddr_t   out;          /*  pointer to results        */
    bool_t  (*eachresult)();/*  call with each result gotten */
```

The `eachresult` procedure is called each time a result is obtained. This procedure returns a Boolean value that indicates whether the caller wants more responses.

```
bool_t done;
    ...
done = eachresult(resultsp, raddr)
    caddr_t resultsp;
    struct sockaddr_in *raddr; /*  Addr of responding machine  */
```

If the `done` parameter returns a value of TRUE, then broadcasting stops and the `clnt_broadcast` routine returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after a few seconds of waiting. If no response comes back, the routine returns with a value of RPC_TIMEDOUT.

Related Information

The **clnt_broadcast** subroutine.

List of RPC Examples on page 7–49.

Understanding the RPC Features on page 7–29, Broadcasting Remote Procedure Calls on page 7–30.

Example Using the Highest Layer of RPC

The following example shows how a program that determines how many users are logged into a remote workstation does so by calling the RPC library **rnusers** routine:

```
#include <stdio.h>

main(argc, argv)
    int argc;
    char **argv;
{
    int num;

            if (argc != 2) {
                  fprintf(stderr, "usage: rnusers hostname\n");
                  exit(1);
            }
            if ((num = rnusers(argv[1])) < 0) {
                   fprintf(stderr, "error: rnusers\n");
                   exit(-1);
            }
            printf("%d users on %s\n", num, argv[1]);
            exit(0);
}
```

Related Information

List of RPC Examples on page 7-49.

Example Using the Intermediate Layer of RPC

The following example shows a simple interface that makes explicit remote procedure calls using the **callrpc** routine at the intermediate layer of RPC.

Intermediate Layer of RPC on the Server Side

Normally, the server registers each procedure, and then goes into an infinite loop waiting to service requests. Since there is only a single procedure to register, the main body of the server message would look like the following:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <utmp.h>
#include <rpcsvc/rusers.h>

char *nuser();

main()
{
    registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
        nuser, xdr_void, xdr_u_long);
    svc_run();        /* Never returns */
    fprintf(stderr, "Error: svc_run returned!\n");
    exit(1);
}
```

The **registerrpc** routine registers a C procedure as corresponding to a given RPC procedure number. The first three parameters, `RUSERPROG`, `RUSERSVERS`, and `RUSERSPROC_NUM`, are the program, version, and procedure numbers of the remote procedure to be registered; the `nuser` parameter is the name of the local procedure that implements the remote procedure; and the `xdr_void` and `xdr_u_long` parameters are the XDR filters for the remote procedure's arguments and results, respectively.

Intermediate Layer of RPC on the Client Side

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <utmp.h>
#include <rpcsvc/rusers.h>

main(argc, argv)
    int argc;
    char **argv;
{
    unsigned long nusers;
    int stat;

    if (argc != 2) {
        fprintf(stderr, "usage: nusers hostname\n");
        exit(-1);
    }
    if (stat = callrpc(argv[1],
        RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
        xdr_void, 0, xdr_u_long, &nusers) != 0) {
            clnt_perrno(stat);
            exit(1);
```

```
    }
    printf("%d users on %s\n", nusers, argv[1]);
    exit(0);
}
```

The **callrpc** subroutine has eight parameters. The first is the name of the remote server machine. The next three parameters are the program, version, and procedure numbers. The fifth and sixth parameters are an XDR filter and an argument to be encoded and passed to the remote procedure. The final two parameters are a filter for decoding the results returned by the remote procedure and a pointer to the place where the procedure's results are to be stored. Multiple arguments and results are handled by embedding them in structures. If the **callrpc** subroutine completes successfully, it returns zero. Otherwise, it returns a nonzero value.

Since data types may be represented differently on different machines, the **callrpc** subroutine needs both the type of the RPC argument and a pointer to the argument itself. For the `RUSERSPROC_NUM` parameter, the return value is unsigned long, so the **callrpc** subroutine has `xdr_u_long` as its first return parameter. This says that the result is of the unsigned long type. The second return parameter, `&nusers`, is a pointer to where the long result is placed. Since the `RUSERSPROC_NUM` parameter takes no argument, the argument parameter of the **callrpc** subroutine is `xdr_void`.

Related Information

The **callrpc** subroutine, **clnt_perrno** subroutine, **registerrpc** subroutine, **svc_run** subroutine, **xdr_u_long** subroutine, **xdr_void** subroutine.

List of RPC Examples on page 7–49.

Using the Intermediate Layer of RPC on page 7–23.

Example Showing How RPC Passes Arbitrary Data Types

The following two examples show how RPC handles arbitrary data types:

Example Passing a Simple User-Defined Structure

```
struct simple {
        int a;
       short b;
} simple;

callrpc(hostname, PROGNUM, VERSNUM, PROCNUM,
        xdr_simple, &simple ...);
```

where the **xdr_simple** function is written as:

```
#include <rpc/rpc.h>

xdr_simple(xdrsp, simplep)
    XDR *xdrsp;
    struct simple *simplep;
{
    if (!xdr_int(xdrsp, &simplep->a))
        return (0);
    if (!xdr_short(xdrsp, &simplep->b))
        return (0);
    return (1);
}
```

Example Passing a Variable-Length Array

```
struct varintarr {
    int *data;
    int arrlnth;
} arr;


callrpc(hostname, PROGNUM, VERSNUM, PROCNUM,
        xdr_varintarr, &arr...);
```

The **xdr_varintarr** subroutine is defined as:

```
xdr_varintarr(xdrsp, arrp)
    XDR *xdrsp;
    struct varintarr *arrp;
{
    return (xdr_array(xdrsp, &arrp->data, &arrp->arrlnth,
        MAXLEN, sizeof(int), xdr_int));
}
```

This routine's parameters are the XDR handle (`xdrsp`), a pointer to the array (`aarp->data`), a pointer to the size of the array (`aarp->arrlnth`), the maximum allowable array size (`MAXLEN`), the size of each array element (`sizeof`), and an XDR routine for handling each array element (`xdr_int`).

Example Passing a Fixed-Length Array

If the size of the array is known in advance, the programmer can call the **xdr_vector** subroutine to serialize fixed-length arrays as in the following example:

```
int intarr[SIZE];

xdr_intarr(xdrsp, intarr)
    XDR *xdrsp;
    int intarr[];
{
    int i;

    return (xdr_vector(xdrsp, intarr, SIZE, sizeof(int),
        xdr_int));
}
```

Example Passing Structure with Pointers

The following example calls the previously written **xdr_simple** routine as well as the built-in **xdr_string** and **xdr_reference** functions. The **xdr_reference** routine chases pointers as shown in the following example:

```
struct finalexample {
    char *string;
    struct simple *simplep;
} finalexample;

xdr_finalexample(xdrsp, finalp)
    XDR *xdrsp;
    struct finalexample *finalp;
{

    if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
        return (0);
    if (!xdr_reference(xdrsp, &finalp->simplep,
     sizeof(struct simple), xdr_simple);
        return (0);
    return (1);
}
```

Related Information

The **callrpc** subroutine, **xdr_array** subroutine, **xdr_int** subroutine, **xdr_reference** subroutine, **xdr_short** subroutine, **xdr_string** subroutine, **xdr_vector** subroutine.

List of RPC Examples on page 7–49.

Programming in RPC on page 7–20, Passing Arbitrary Data Types on page 7–25.

Example Using the Lowest Layer of RPC

The following is an example of the lowest layer of RPC on the server and client side using an **nusers** program.

The Lowest Layer of RPC from the Server Side

The server for the **nusers** program in the following example does the same thing as a program using the **registerrpc** subroutine at the higher level of RPC, but the following is written using a lower layer of the RPC package:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <utmp.h>
#include <rpcsvc/rusers.h>

main()
{
    SVCXPRT *transp;
    int nuser();

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL){
        fprintf(stderr, "can't create an RPC server\n");
        exit(1);
    }
    pmap_unset(RUSERSPROG, RUSERSVERS);
    if (!svc_register(transp, RUSERSPROG, RUSERSVERS,

              nuser, IPPROTO_UDP)) {
        fprintf(stderr, "can't register RUSER service\n");
        exit(1);
    }
    svc_run();  /*  Never returns  */
    fprintf(stderr, "should never reach this point\n");
}

    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0))
            fprintf(stderr, "can't reply to RPC call\n");
        return;
    case RUSERSPROC_NUM:
        /*
         * Code here to compute the number of users
         * and assign it to the nusers variable
         */
        if (!svc_sendreply(transp, xdr_u_long, &nusers))
            fprintf(stderr, "can't reply to RPC call\n");
        return;
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

First, the server gets a transport handle, which is used for receiving and replying to RPC messages. The **registerrpc** routine calls the **svcudp_create** routine to get a UDP handle. If a more reliable protocol is required, the **svctcp_create** routine can be called instead. If the argument to the **svcudp_create** routine is RPC_ANYSOCK, the RPC library creates a socket on which to receive and reply to remote procedure calls. Otherwise, the **svcudp_create** routine expects its argument to be a valid socket number. If the programmer specifies his own socket, it can be bound or unbound. If it is bound to a port by the programmer, the port numbers of the **svcudp_create** routine and the **clnttcp_create** routine (the low-level client routine) must match.

If the programmer specifies the RPC_ANYSOCK argument, the RPC library routines open sockets. The **svcudp_create** and **clntudp_create** routines cause the RPC library routines to bind the appropriate socket if it is not already bound.

A service may register its port number with the local port mapper service. This is done by specifying a non-zero protocol number in the **svc_register** routine. A programmer at the client machine can discover the server port number by consulting the port mapper at the server workstation. This is done automatically by specifying a zero port number in the **clntudp_create** or **clnttcp_create** routines.

After creating a service transport (SVCXPRT) handle, the next step is to call the **pmap_unset** routine. If the **nusers** server crashed earlier, this routine erases any trace of it before restarting. Specifically, the **pmap_unset** routine erases the entry for RUSERSPROG from the port mapper's tables.

Finally, the program number for nusers is associated with the **nuser** procedure. The final argument to the **svc_register** routine is normally the protocol being used, which in this case is IPPROTO_UDP. Registration is performed at the program level, rather than the procedure level.

The **nuser** user service routine must call and dispatch the appropriate XDR routines based on the procedure number. The **nuser** routine requires two tasks, unlike the **registerrpc** routine which performs them automatically. The first is that the NULLPROC procedure (currently zero) returns with no results. This can be used as a simple test for detecting whether a remote program is running. Second, there is a check for invalid procedure numbers. If one is detected, the **svcerr_noproc** routine is called to handle the error.

The user service routine serializes the results and returns them to the RPC caller through the **svc_sendreply** routine. The first parameter of this routine is the SVCXPRT handle, the second is the XDR routine that indicates return data type, and the third is a pointer to the data to be returned.

As an example, a RUSERSPROC_BOOL procedure can be added, which has an **nusers** argument and returns a value of TRUE or FALSE, depending on whether there are nusers logged on. The following example shows this addition:

```
case RUSERSPROC_BOOL: {
    int bool;
    unsigned nuserquery;

    if (!svc_getargs(transp, xdr_u_int, &nuserquery) {
        svcerr_decode(transp);
        return;
    }
    /*
     * Code to set nusers = number of users
     */
    if (nuserquery == nusers)
        bool = TRUE;
    else
        bool = FALSE;
    if (!svc_sendreply(transp, xdr_bool, &bool)) {
         fprintf(stderr, "can't reply to RPC call\n");
        return (1);
    }
    return;
}
```

The **svc_getargs** routine takes the following arguments: an SVCXPRT handle, the XDR routine, and a pointer that indicates where to place the input.

The Lowest Layer of RPC from the Client Side

When a programmer uses the **callrpc** routine, there is no control over the RPC delivery mechanism or the socket used to transport the data. To illustrate the lowest layer of RPC, which allows the user to adjust these parameters, the following code can be used to request the **nusers** service:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <utmp.h>
#include <rpcsvc/rusers.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

main(argc, argv)
    int argc;
    char **argv;
{
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int sock = RPC_ANYSOCK;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    unsigned long nusers;
```

```
    if (argc != 2) {
        fprintf(stderr, "usage: nusers hostname\n");
        exit(-1);
    }
    if ((hp = gethostbyname(argv[1])) == NULL) {
        fprintf(stderr, "can't get addr for %s\n",argv[1]);
        exit(-1);
    }
    pertry_timeout.tv_sec = 3;
    pertry_timeout.tv_usec = 0;
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
        hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port =  0;
    if ((client = clntudp_create(&server_addr, RUSERSPROG,
     RUSERSVERS, pertry_timeout, &sock)) == NULL) {
        clnt_pcreateerror("clntudp_create");
        exit(-1);
    }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
     clnt_stat = clnt_call(client, RUSERSPROC_NUM, xdr_void,
        0, xdr_u_long, &nusers, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "rpc");
        exit(-1);
    }
    clnt_destroy(client);
    close(sock);
    exit(0);
}
```

The low-level version of the **callrpc** routine is the **clnt_call** macro, which takes a `CLIENT` pointer rather than a host name. The parameters to the **clnt_call** macro are a `CLIENT` pointer, the procedure number, the XDR routine for serializing the argument, a pointer to the argument, the XDR routine for deserializing the return value, a pointer to where the return value is to be placed, and the total time in seconds to wait for a reply. Thus, the number of tries is the **clnt_call** time out divided by the **clntudp_create** timeout.

The `CLIENT` pointer is encoded with the transport mechanism. The **callrpc** routine uses UDP, thus it calls the **clntudp_create** routine to get a `CLIENT` pointer. To get TCP (Transmission Control Protocol), the programmer can call the **clnttcp_create** routine.

The parameters to the **clntudp_create** routine are the server address, the program number, the version number, a time out value (between tries), and a pointer to a socket.

The **clnt_destroy** call always deallocates the space associated with the client handle. If the RPC library opened the socket associated with the client handle, the **clnt_destroy** macro closes it. If the socket was opened by the programmer, it stays open. In cases where there are multiple client handles using the same socket, it is possible to destroy one handle without closing the socket that other handles are using.

The stream connection is made when the call to the **clntudp_create** macro is replaced by a call to the **clnttcp_create** routine.

```
clnttcp_create(&server_addr, prognum, versnum, &sock,
               inputsize, outputsize);
```

In following example, there is no timeout argument. Instead, the send and receive buffer sizes must be specified. When the **clnttcp_create** call is made, a TCP connection is established. All remote procedure calls using the client handle use the TCP connection. The server side of a remote procedure call using TCP is similar, except that the **svcudp_create** routine is replaced by the **svctcp_create** routine, as follows:

```
transp = svctcp_create(RPC_ANYSOCK, 0, 0);
```

The last two arguments to the `svctcp_create` routine are send and receive sizes respectively. If 0 is specified for either of these, the system chooses a reasonable default.

Related Information

The **clnt_call** macro, **clnt_destroy** macro, **svc_getargs** macro.

The **callrpc** subroutine, **clnt_pcreateerror** subroutine, **clnt_perror** subroutine, **clnttcp_create** subroutine, **clntudp_create** subroutine, **pmap_unset** subroutine, **registerrpc** subroutine, **svcerr_decode** subroutine, **svcerr_noproc** subroutine, **svc_register** subroutine, **svc_run** subroutine, **svc_sendreply** subroutine, **svctcp_create** subroutine, **svcudp_create** subroutine, **xdr_bool** subroutine, **xdr_u_long** subroutine, **xdr_void** subroutine.

List of RPC Examples on page 7–49.

Example Using the select Subroutine

The code for the **svc_run** routine with the **select** subroutine is as follows:

```
void
svc_run()
{
    fd_set readfds;
    int dtbsz = getdtablesize();

    for (;;) {
        readfds = svc_fds;
        switch (select(dtbsz, &readfds, NULL,NULL,NULL)) {

        case -1:
            if (errno == EINTR)
                continue;
            perror("select");
            return;
        case 0:
            break;
        default:
            svc_getreqset(&readfds);
        }
    }
}
```

Related Information

The **select** subroutine, **svc_getreqset** subroutine, **svc_run** subroutine.

List of RPC Examples on page 7–49.

Understanding the RPC Features on page 7–29.

Example Using rcp on TCP

The following is an example using **rcp**. This example also illustrates an XDR procedure that behaves differently on serialization than on deserialization. The initiator of the RPC **snd** call takes its standard input and sends it to the server **rcv** process, which prints it on standard output. The **snd** call uses Transmission Control Protocol (TCP).

The routine follows:

```
/*
 * The xdr routine:
 *       on decode, read from wire, write onto fp
 *       on encode, read from fp, write onto wire
 */
#include <stdio.h>
#include <rpc/rpc.h>

xdr_rcp(xdrs, fp)
      XDR *xdrs;
      FILE *fp;

{
      unsigned long size;
      char buf[BUFSIZ], *p;
      if (xdrs->x_op == XDR_FREE)/* nothing to free */
             return 1;
      while (1) {
             if (xdrs->x_op == XDR_ENCODE) {
                    if ((size = fread(buf, sizeof(char), BUFSIZ,
                      fp)) == 0 && ferror(fp)) {
                           fprintf(stderr, "can't fread\n");
                           return (1);
                    }
             }
             p = buf;
             if (!xdr_bytes(xdrs, &p, &size, BUFSIZ))
                    return 0;
             if (size == 0)
                    return 1;
             if (xdrs->x_op == XDR_DECODE) {
                    if (fwrite(buf, sizeof(char),size,
                      fp) != size) {
                           fprintf(stderr, "can't fwrite\n");
                           return (1);
                    }
             }
      }

}

/*
 * The sender routines
 */
#include <stdio.h>
#include <netdb.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>
```

```
main(argc, argv)
      int argc;
      char **argv;
{
      int xdr_rcp();
      int err;
      if (argc < 2) {
             fprintf(stderr, "usage: %s servername\n", argv[0]);
             exit(-1);
      }
      if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC,
        RCPVERS, xdr_rcp, stdin, xdr_void, 0) != 0)) {
             clnt_perrno(err);
             fprintf(stderr, "can't make RPC call\n");
             exit(1);
      }
      exit(0);
}

callrpctcp(host, prognum, procnum, versnum, inproc, in,
            outproc, out)
      char *host, *in, *out;
      xdrproc_t inproc, outproc;
{
      struct sockaddr_in server_addr;
      int socket = RPC_ANYSOCK;
      enum clnt_stat clnt_stat;
      struct hostent *hp;
      register CLIENT *client;
      struct timeval total_timeout;

      if ((hp = gethostbyname(host)) == NULL) {
             fprintf(stderr, "can't get addr for '%s'\n", host);
             return (-1);
      }
      bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
             hp->h_length);
      server_addr.sin_family = AF_INET;
      server_addr.sin_port =  0;
      if ((client = clnttcp_create(&server_addr, prognum,
        versnum, &socket, BUFSIZ, BUFSIZ)) == NULL) {
             perror("rpctcp_create");
             return (-1);
      }
      total_timeout.tv_sec = 20;
      total_timeout.tv_usec = 0;
      clnt_stat = clnt_call(client, procnum,
             inproc, in, outproc, out, total_timeout);
      clnt_destroy(client);
      return (int)clnt_stat;
}
/*
 * The receiving routines
 */
#include <stdio.h>
#include <rpc/rpc.h>
```

```
main()
{
      register SVCXPRT *transp;
      int rcp_service(), xdr_rcp();

      if ((transp = svctcp_create(RPC_ANYSOCK,
        BUFSIZ, BUFSIZ)) == NULL) {
              fprintf("svctcp_create: error\n");
              exit(1);
      }
      pmap_unset(RCPPROG, RCPVERS);
      if (!svc_register(transp,
        RCPPROG, RCPVERS, rcp_service, IPPROTO_TCP)) {
              fprintf(stderr, "svc_register: error\n");
              exit(1);
      }
      svc_run();  /*  never returns  */
      fprintf(stderr, "svc_run should never return\n");
}

rcp_service(rqstp, transp)
      register struct svc_req *rqstp;
      register SVCXPRT *transp;
{
      switch (rqstp->rq_proc) {
      case NULLPROC:
              if (svc_sendreply(transp, xdr_void, 0) == 0) {
                      fprintf(stderr, "err: rcp_service");
                      return (1);
              }
              return;
      case RCPPROC_FP:
              if (!svc_getargs(transp, xdr_rcp, stdout)) {
                      svcerr_decode(transp);
                      return;
              }
              if (!svc_sendreply(transp, xdr_void, 0)) {
                      fprintf(stderr, "can't reply\n");
                      return;
              }
              return (0);
      default:
              svcerr_noproc(transp);
              return;
      }
}
```

Related Information

The **clnt_call** macro, **clnt_destroy** macro, **svc_getargs** macro.

The **clnt_perrno** subroutine, **clnttcp_create** subroutine, **pmap_unset** subroutine, **svcerr_decode** subroutine, **svcerr_noproc** subroutine, **svc_register** subroutine, **svc_run** subroutine, **svc_sendreply** subroutine, **svctcp_create** subroutine, **xdr_bytes** subroutine, **xdr_free** subroutine, **xdr_void** subroutine.

List of RPC Examples on page 7–49.

Understanding the RPC Features on page 7–29.

Example Using Multiple Program Versions

By convention, the first version number of program PROG is referred to as PROGVERS_ORIG and the most recent version is PROGVERS. For example, the programmer can create a new version of the **user** program that returns an unsigned short value rather than a long value. If the programmer names this version RUSERSVERS_SHORT, then the following program permits the server to support both programs:

```
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_ORIG,
  nuser, IPPROTO_TCP)) {
    fprintf(stderr, "can't register RUSER service\n");
    exit(1);
}
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_SHORT,
  nuser, IPPROTO_TCP)) {
    fprintf(stderr, "can't register RUSER service\n");
    exit(1);
}
```

Both versions can be handled by the same C procedure. This is illustrated in the following example using the **nusers** procedure:

```
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;
    unsigned short nusers2;

    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can't reply to RPC call\n");
                  return (1);
        }
        return;
    case RUSERSPROC_NUM:
        /*
         * Code here to compute the number of users
         * and assign it to the variable nusers
         */
```

```
                nusers2 = nusers;
                switch (rqstp->rq_vers) {
                case RUSERSVERS_ORIG:
                    if (!svc_sendreply(transp, xdr_u_long,
                    &nusers)) {
                        fprintf(stderr,"can't reply to RPC call\n");
                    }
                    break;
                case RUSERSVERS_SHORT:
                    if (!svc_sendreply(transp, xdr_u_short,
                    &nusers2)) {
                        fprintf(stderr,"can't reply to RPC call\n");
                    }
                    break;
                }
            default:
                svcerr_noproc(transp);
                return;
            }
        }
```

Related Information

The **svcerr_noproc** subroutine, **svc_register** subroutine, **svc_sendreply** subroutine, **xdr_u_short** subroutine, **xdr_void** subroutine.

List of RPC Examples on page 7–49.

Programming in RPC on page 7–20, Assigning Version Numbers on page 7–21.

Example Converting Local Procedures into Remote Procedures

This example illustrates one way to convert an application that runs on a single machine into one that runs over a network. For instance, the programmer first creates a program that prints a message to the console, as follows:

```
/*
 * printmsg.c: print a message on the console
 */
#include <stdio.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    char *message;

   if (argc < 2) {
       fprintf(stderr, "usage: %s <message>\n",
            argv[0]);
       exit(1);
    }
    message = argv[1];

    if (!printmessage(message)) {
         fprintf(stderr, "%s: couldn't print your
         message\en", argv[0]);
         exit(1);
    }
    printf("Message Delivered!\n");
    exit(0);
}
/*
 * Print a message to the console.
 * Return a boolean indicating whether the
 *message was actually printed.
 */
printmessage(msg)
    char *msg;
{
    FILE *f;
    f = fopen("/dev/console", "w");
    if (f == NULL) {
        return (0);
    }
    fprintf(f, "%s\en", msg);
    fclose(f);
    return(1);
}
```

The reply message follows:

```
example%  cc printmsg.c —o printmsg
example%  printmsg "Hello, there."
Message delivered!
example%
```

If the `printmessage` program is turned into a remote procedure, it can be called from anywhere in the network. Ideally, one would insert a keyword such as `remote` in front of a procedure to turn it into a remote procedure. Unfortunately, the constraints of the C language do not permit this. However, a procedure can be made remote without language support

To do this, the programmer must know the data types of all procedure inputs and outputs. In this case, the `printmessage` procedure takes a string as input and returns an integer as output. Knowing this, the programmer can write a protocol specification in RPC language that describes the remote version of `printmessage`, as follows:

```
/*
 * msg.x: Remote message printing protocol
 */
program MESSAGEPROG {
   version MESSAGEVERS {
       int PRINTMESSAGE(string) = 1;
   } = 1;
} = 99;
```

Remote procedures are part of remote programs, so the above protocol declares a remote program which contains the single procedure `PRINTMESSAGE`. This procedure was declared to be in version 1 of the remote program. No null procedure (procedure 0) is necessary because the **rpcgen** command generates it automatically.

Conventionally, all declarations are written with capital letters.

The argument type is `string` and not `char *` because a `char *` in C is ambiguous. Programmers usually intend it to mean a null-terminated string of characters, but it could also represent a pointer to a single character or a pointer to an array of characters. In RPC language, a null-terminated string is unambiguously called a string.

Next, the programmer writes the remote procedure itself. The definition of a remote procedure to implement the `PRINTMESSAGE` procedure declared above can be written as follows:

```
/*
 * msg_proc.c: implementation of the remote
 *procedure "printmessage"
 */
#include <stdio.h>
#include <rpc/rpc.h>    /* always needed  */
#include "msg.h"    /* msg.h will be generated by rpcgen */
/*
 * Remote version of "printmessage"
 */ int *
```

```
printmessage_1(msg)
    char **msg;
{
    static int result;  /* must be static! */
    FILE *f;

    f = fopen("/dev/console", "w");
    if (f == NULL) {
        result = 0;
        return (&result);
    }
    fprintf(f, "%s\en", *msg);
    fclose(f);
    result = 1;
    return (&result);
}
```

The declaration of the remote procedure `printmessage_1` in this step differs from that of the local procedure `printmessage` in the first step in three ways:

- It takes a pointer to a string instead of a string itself. This is true of all remote procedures, which always take pointers to their arguments rather than the arguments themselves.
- It returns a pointer to an integer instead of the integer itself. This is also true of remote procedures, which generally return a pointer to their results.
- It has a `_1` appended to its name. Remote procedures called by the **rpcgen** command are named by the following rule: the name in the program definition (here `PRINTMESSAGE`) is converted to all lower-case letters, and an _ (underbar) and the version number are appended.

Finally, the programmers declare the main client program that will call the remote procedure, as follows:

```
/*
 * rprintmsg.c: remote version of "printmsg.c"
 */
#include <stdio.h>
#include <rpc/rpc.h>      /* always needed */
#include "msg.h"  /* msg.h will be generated by rpcgen */

main(argc, argv)
    int argc;
    char *argv[];
{
    CLIENT *cl;
    int *result;
    char *server;
    char *message;

    if (argc < 3) {
        fprintf(stderr,
        "usage: %s host message\en", argv[0]);
        exit(1);
    }
    /*
     * Save values of command line arguments
     */
```

```
    server = argv[1];
    message = argv[2];

    /*
     * Create client "handle" used for calling MESSAGEPROG on
     * the server designated on the command line.  We tell
     * the RPC package to use the "tcp" protocol when
     * contacting the server.
     */
    cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");
    if (cl == NULL) {
         /*
          * Couldn't establish connection with server.
          * Print error message and die.
          */
          clnt_pcreateerror(server);
          exit(1);
    }
   /*
    * Call the remote procedure "printmessage" on the server
    */
    result = printmessage_1(&message, cl);
    if (result == NULL) {
        /*
         * An error occurred while calling the server.
         * Print error message and die.
         */
        clnt_perror(cl, server);
        exit(1);
    }
    /*
     * Okay, we successfully called the remote procedure.
     */
    if (*result == 0) {
        /*
         * Server was unable to print our message.
         * Print error message and die.
         */
    fprintf(stderr, "%s: %s couldn't print your message\n",
        argv[0], server);
        exit(1);
    }
    /*
     * The message got printed on the server's console
     */
     printf("Message delivered to %s!\n", server);
     exit(0);
}
```

There are two things to note here:

1. First a client handle is created using the RPC library routine `clnt_create`. This client handle is passed to the stub routines that call the remote procedure.
2. The remote procedure `printmessage_1` is called exactly the same way as it is declared in `msg_proc.c`, except for the inserted client handle as the first argument.

The client program rprintmsg and the server program msg_server are compiled as follows:

```
example%  rpcgen msg.x
example%  cc rprintmsg.c msg_clnt.c –o rprintmsg
example%  cc msg_proc.c msg_svc.c –o msg_server
```

Before compilation, however, rpcgen is used to perform the following operations on the input file msg.x:

- It creates a header file called **msg.h** that contains #defines for MESSAGEPROG, MESSAGEVERS, and PRINTMESSAGE for use in the other modules.
- It creates a client stub routine in the msg_clnt.c file. In this case there is only one, the printmessage_1 that is referred to from the printmsg client program. The name of the output file for client stub routines is always formed in this way: if the name of the input file is **FOO.x**, the client stub's output file is called **FOO_clnt.c**.
- It creates the server program which calls printmessage_1 in msg_proc.c. This server program is named msg_svc.c. The rule for naming the server output file is similar to the previous one: for an input file called **FOO.x**, the output server file is named **FOO_svc.c**.

Related Information

The **clnt_create** subroutine, **clnt_pcreateerror** subroutine, **clnt_perror** subroutine.

List of RPC Examples on page 7–49.

Understanding the rpcgen Protocol Compiler on page 7–37, Converting Local Procedures into Remote Procedures on page 7–38.

Example Generating XDR Routines

The previous example demonstrates the automatic generation of client and server RPC code. The **rpcgen** protocol compiler may also be used to generate XDR routines, that is, the routines necessary to convert local data structures into network format and vice versa. The following example presents a complete RPC service—a remote directory listing service that uses the **rpcgen** protocol compiler not only to generate stub routines, but also to generate XDR routines. Here is the protocol description file:

```
/*
 * dir.x: Remote directory listing protocol
 */
const MAXNAMELEN = 255;/* maximum length of a directory entry    */
typedef string nametype<MAXNAMELEN>;     /* a directory entry    */
typedef struct namenode *namelist;    /* a link in the listing    */
/*
 * A node in the directory listing
 */
struct namenode {
     nametype name;      /* name of directory entry              */
     namelist next;      /* next entry                           */
};
/*
 * The result of a READDIR operation.
 */
 union readdir_res switch (int errno) {
 case 0:
     namelist list;  /* no error: return directory listing       */
 default:
     void;        /* error occurred: nothing else to return      */
};
/*
 * The directory program definition
 */
program DIRPROG {
     version DIRVERS {
             readdir_res
             READDIR(nametype) = 1;
         } = 1;
} = 76;
```

Note: Types (like **readdir_res** in the example above) can be defined using the `struct`, `union` and `enum` keywords, but those keywords should not be used in subsequent declarations of variables of those types. For example, if you define a union `foo`, you should declare using only `foo` and not `union foo`. In fact, the **rpcgen** protocol compiler compiles RPC unions into C structures, and it is an error to declare them using the `union` keyword.

Running the **rpcgen** protocol compiler on **dir.x** creates four output files. Three are the same as before: header file, client stub routines, and server skeleton. The fourth file contains the XDR routines necessary for converting the specified data types into XDR format and vice versa. These are output in the **dir_xdr.c** file.

The following is the implementation of the READDIR procedure:

```
/*
 * dir_proc.c: remote readdir implementation
 */
#include <rpc/rpc.h>
#include <sys/dir.h>
#include "dir.h"

extern int errno;
extern char *malloc();
extern char *strdup();

readdir_res *
readdir_1(dirname)
        nametype *dirname;
{
        DIR *dirp;
        struct direct *d;
        namelist nl;
        namelist *nlp;
        static readdir_res res; /* must be static                 */

        /*
         * Open directory
         */
        dirp = opendir(*dirname);
        if (dirp == NULL) {
                res.errno = errno;
                return (&res);
        }
        /*
         * Free previous result
         */
        xdr_free(xdr_readdir_res, &res);
        /*
         * Collect directory entries.
         * Memory allocated here will be freed by xdr_free
         * next time readdir_1 is called
         */
        nlp = &res.readdir_res_u.list;
        while (d = readdir(dirp)) {
           nl = *nlp = (namenode *) malloc(sizeof(namenode));
           nl->name = strdup(d->d_name);
           nlp = &nl->next;
        }
         *nlp = NULL;

         /*
         * Return the result
         */
         res.errno = 0;
        closedir(dirp);
        return (&res);
}
```

Finally, there is the client side program to call the server:

```
/*
 * rls.c: Remote directory listing client
 */
 #include <stdio.h>
 #include <rpc/rpc.h>     /* always need this                         */
 #include "dir.h"         /* will be generated by rpcgen              */

 extern int errno;

 main(argc, argv)
         int argc;
         char *argv[];
 {
         CLIENT *cl;
         char *server;
         char *dir;
         readdir_res *result;
         namelist nl;

         if (argc != 3) {
                fprintf(stderr, "usage: %s host directory\n",
                   argv[0]);
                 exit(1);
         }
         /*
          * Remember what our command line arguments refer to
          */
         server = argv[1];
         dir = argv[2];

         /*
          * Create client "handle" used for calling MESSAGEPROG on
          * the server designated on the command line. We tell the
          *  RPC package to use the "tcp" protocol when contacting
          * the server.
          */
         cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");
         if (cl == NULL) {
              /*
               * Couldn't establish connection with server.
               * Print error message and die.
               */
               clnt_pcreateerror(server);
               exit(1);
         }
         /*
          * Call the remote procedure readdir on the server
           */
         result = readdir_1(&dir, cl);
         if (result == NULL) {
                 /*
                  * An error occurred while calling the server.
                  * Print error message and die.
                  */
                 clnt_perror(cl, server);
                 exit(1);
```

```
        }
        /*
         * Okay, we successfully called the remote procedure.
         */
        if (result->errno != 0) {
                /*
                 * A remote system error occurred.
                 * Print error message and die.
                 */
                errno = result->errno;
                perror(dir);
                exit(1);
        }
        /*
         * Successfully got a directory listing.
         * Print it out.
         */
        for (nl = result->readdir_res_u.list; nl != NULL;
          nl = nl->next) {
                printf("%s\en", nl->name);
        }
        exit(0);
}
```

A final note about the **rpcgen** protocol compiler: The client program and the server procedure can be tested together as a single program by simply linking them with each other rather than with the client and server stubs. The procedure calls are executed as ordinary local procedure calls and the program can be debugged with a local debugger such as dbx. When the program is working, the client program can be linked to the client stub produced by the **rpcgen** protocol compiler and the server procedures can be linked to the server stub produced by the **rpcgen** protocol compiler.

Note: If you do this, you may want to comment out calls to RPC library routines and have client-side routines call server routines directly.

Related Information

List of RPC Examples on page 7–49.
Understanding the rpcgen Protocol Compiler on page 7–37.

Example Using RPC Callback Procedures

Occasionally, it is useful to have a server become a client and make a remote procedure call back to the process that is its client. For example, with remote debugging, the client is a window system program and the server is a debugger running on the remote machine. Usually, the user clicks a mouse button at the debugging window, which converts this to a debugger command, and then makes a remote procedure call to the server (where the debugger is actually running), telling it to execute that command. However, when the debugger hits a breakpoint, the roles are reversed, and the debugger makes a remote procedure call to the window program to inform the user that a breakpoint has been reached.

To do an RPC callback, you need a program number to make the remote procedure call on. Since this is a dynamically generated program number, it should be in the transient range, `0x40000000 to 0x5fffffff`. The **gettransient** routine returns a valid program number in the transient range, and registers it with the port mapper. This routine only talks to the port mapper running on the same machine as the **gettransient** routine itself. The call to the **pmap_set** routine is a test-and-set operation. That is, it indivisibly tests whether a program number has already been registered, and reserves the number if not. On return, the **sockp** argument contains a socket that can be used as the argument to an **svcudp_create** or **svctcp_create** routine.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>

gettransient(proto, vers, sockp)
      int proto, vers, *sockp;
{
      static int prognum = 0x40000000;
      int s, len, socktype;
      struct sockaddr_in addr;

      switch(proto) {
              case IPPROTO_UDP:
                      socktype = SOCK_DGRAM;
                      break;
              case IPPROTO_TCP:
                      socktype = SOCK_STREAM;
                      break;
              default:
                      fprintf(stderr, "unknown protocol type\n");
                      return 0;
```

```
        }
        if (*sockp == RPC_ANYSOCK) {
             if ((s = socket(AF_INET, socktype, 0)) < 0) {
                  perror("socket");
                  return (0);
             }
             *sockp = s;
        }
        else
             s = *sockp;
        addr.sin_addr.s_addr = 0;
        addr.sin_family = AF_INET;
        addr.sin_port = 0;
        len = sizeof(addr);
       /*
         * may be already bound, so don't check for error
         */
       bind(s, &addr, len);
        if (getsockname(s, &addr, &len)< 0) {
             perror("getsockname");
             return (0);
        }
        while (!pmap_set(prognum++, vers, proto,
             ntohs(addr.sin_port))) continue;
        return (prognum-1);
}
```

Note: The call to the **ntohs** subroutine is necessary to ensure that the port number in `addr.sin_port`, which is in network byte order, is passed in host byte order (as the **pmap_set** subroutine expects).

The following pair of programs illustrate how to use the **gettransient** routine. The client makes an remote procedure call to the server, passing it a transient program number. Then the client waits around to receive a callback from the server at that program number. The server registers the program EXAMPLEPROG so that it can receive the remote procedure call informing it of the callback program number. Then at some randomly selected time (on receiving an ALRM signal in this example), the server sends a callback remote procedure call, using the program number it received earlier.

```
/*
 * client
 */

#include <stdio.h>
#include <rpc/rpc.h>

int callback();
char hostname[256];

main()
{
```

```
        int x, ans, s;
        SVCXPRT *xprt;
        gethostname(hostname, sizeof(hostname));
        s = RPC_ANYSOCK;
        x = gettransient(IPPROTO_UDP, 1, &s);
        fprintf(stderr, "client gets prognum %d\n", x);
        if ((xprt = svcudp_create(s)) == NULL) {
          fprintf(stderr, "rpc_server: svcudp_create\n");
             exit(1);
        }
        /* protocol is 0 - gettransient does registering
         */
        (void)svc_register(xprt, x, 1, callback, 0);
        ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEVERS,
             EXAMPLEPROC_CALLBACK, xdr_int, &x, xdr_void, 0);
        if ((enum clnt_stat) ans != RPC_SUCCESS) {
             fprintf(stderr, "call: ");
             clnt_perrno(ans);
             fprintf(stderr, "\n");
        }
        svc_run();
        fprintf(stderr, "Error: svc_run shouldn't return\n");
}

callback(rqstp, transp)
        register struct svc_req *rqstp;
        register SVCXPRT *transp;
{
        switch (rqstp->rq_proc) {
             case 0:
                  if (!svc_sendreply(transp, xdr_void, 0)) {
                       fprintf(stderr, "err: exampleprog\n");
                       return (1);
                  }
                  return (0);
             case 1:
                  if (!svc_getargs(transp, xdr_void, 0)) {
                       svcerr_decode(transp);
                       return (1);
                  }
                  fprintf(stderr, "client got callback\n");
                  if (!svc_sendreply(transp, xdr_void, 0)) {
                       fprintf(stderr, "err: exampleprog");
                       return (1);
                  }
        }
}
/*
 * server
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>
```

```
char *getnewprog();
char hostname[256];
int docallback();
int pnum;           /*  program number for callback routine  */

main()

{
      gethostname(hostname, sizeof(hostname));
      registerrpc(EXAMPLEPROG, EXAMPLEVERS,
        EXAMPLEPROC_CALLBACK, getnewprog, xdr_int, xdr_void);
      fprintf(stderr, "server going into svc_run\n");
      signal(SIGALRM, docallback);
      alarm(10);
      svc_run();
      fprintf(stderr, "Error: svc_run shouldn't return\n");
}

char *
getnewprog(pnump)
     char *pnump;
{
     pnum = *(int *)pnump;
     return NULL;
}

docallback()
{
     int ans;

     ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0,
          xdr_void, 0);
     if (ans != 0) {
          fprintf(stderr, "server: ");
          clnt_perrno(ans);
          fprintf(stderr, "\n");
     }
}
```

Related Information

List of RPC Examples on page 7–49.

Understanding the RPC Features on page 7–29.

Chapter 8. AIX SNA Services/6000

The following information can be used to design and code application programs that use the AIX SNA Services/6000 programming interface to send and receive data over a network. Included in this chapter is information about the AIX Operating System and AIX Library subroutines that are part of the programming interface to AIX SNA Services/6000. Also included is the content of the header files used with the programming interface to AIX SNA Services/6000.

There is information about the generic SNA device driver, which provides support that allows the generic SNA application code to use the PU Services of AIX SNA Services/6000. The SNA Services/6000 LU0Facility is explained and the subsystem is defined. The application program interface is also defined in this chapter.

Finally four transaction program examples, a file transfer program description, and information for writing a generic SNA application are supplied.

AIX SNA Services/6000 Subroutines

AIX SNA Services/6000 Subroutines describes two kinds of subroutines: (1) the AIX SNA Services/6000 Operating System subroutines that you can use from a program to control the operation of AIX SNA Services/6000 and, (2) the AIX SNA Services/6000 Library subroutines that you can use in your application program to control the general operation of AIX SNA Services/6000 and send network management information to the host.

AIX SNA Services/6000 Operating System Subroutines

The AIX SNA Services/6000 Operating System subroutines are:

- **open** subroutine for SNA Services/6000
- **close** subroutine for SNA Services/6000
- **read** subroutine for SNA Services/6000
- **write** subroutine for SNA Services/6000
- **ioctl** subroutine for SNA Services/6000
- **readx** subroutine for SNA Services/6000
- **writex** subroutine for SNA Services/6000
- **select** subroutine for SNA Services/6000.

Refer to the Subroutines Overview in *Communications Programming Concepts* for general reference information about the syntax and functions provided by these subroutines. This document discusses their use with AIX SNA Services/6000 only.

AIX SNA Services/6000 Operating System Subroutine Interfaces

AIX SNA Services/6000 has two subroutine interfaces: limited and extended. Both of these interfaces use standard AIX SNA Services/6000 Operating System subroutines. When you configure a network, you must specify in the connection profile which subroutine interface the transaction program uses (refer to the discussion of the connection profile in Defining Remote Connection Characteristics in *Communication Concepts and Procedures*). You can use either interface to write an application transaction program that issues routines directly to the SNA device driver. In addition, you can use the extended interface to write kernel transaction programs, combine functions into one routine, or both.

Be sure to document which interface you choose and pass this information on to the person who installs your program. The connection profile requires this information to install your program (see Defining Remote Connection Characteristics in *Communication Concepts and Procedures*).

Limited Interface

The limited interface allows application programs to perform basic data send and receive operations for an LU 6.2 protocol boundary only. It does not provide any complex send and receive operations or monitor network status. It cannot be used for LUs 1, 2, or 3. The subroutines that implement this interface are:

- **open** subroutine for SNA Services/6000
- **close** subroutine for SNA Services/6000
- **read** subroutine for SNA Services/6000
- **write** subroutine for SNA Services/6000.

Programs that use the limited interface must observe the following restrictions:

- The session must be for LU 6.2.
- Only one conversation can be allocated for each open connection.
- The conversation must be a basic conversation.
- No additional information other than data can be sent or received.

In this interface, the resource ID is not used. Likewise, the limited interface does not support PIP data. The profile pointer (file descriptor) is returned by calling the **open** subroutine in the transaction program. See Remote Transaction Program Example Program on page 8–48 .

Extended Interface

The extended interface allows a kernel or application transaction program to control AIX SNA Services/6000 operations by directly communicating with the SNA device driver. You can use this interface to control operations for LUs 1, 2, 3, or 6.2, and to allocate more than one conversation for an open connection. This interface uses the following AIX SNA Services/6000 Operating System subroutines:

- **open** subroutine for SNA Services/6000
- **close** subroutine for SNA Services/6000
- **select** subroutine for SNA Services/6000
- **read** subroutine for SNA Services/6000
- **readx** subroutine for SNA Services/6000
- **write** subroutine for SNA Services/6000
- **writex** subroutine for SNA Services/6000
- **ioctl** subroutine for SNA Services/6000.

To pass additional information between the SNA device driver and the transaction program, the **readx** and **writex** subroutines use a data structure of type **ext_io_str**. The structure is defined in the **luxsna.h** include file, and the routines pass a pointer to the structure as part of the syntax. Refer to the description of each routine for information about how each uses this structure.

AIX SNA Services/6000 Library Subroutines

The AIX SNA Services/6000 Library subroutines are divided into two groups, the AIX SNA Services/6000 Library Subroutines for Transaction Program Conversations and the AIX SNA Services/6000 Library Subroutines for Network Management.

AIX SNA Services/6000 Library Subroutines for Transaction Program Conversations

AIX SNA Services/6000 provides a set of run-time library subroutines to use in a program to:

- Establish a conversation with one or more remote transaction programs
- Exchange data with that program
- Disconnect from the remote transaction program.

These subroutines are contained in a library file, **/usr/lib/libsna.a**. To use these subroutines in a program, compile the program, using the **–lsna** flag with the **cc** command. For example, to compile the `mytpn.c` program that contains routines to the AIX SNA Services/6000 subroutines, use the following command:

```
cc mytpn.c –o mytpn –lsna
```

The subroutines support two types of SNA conversations:

basic conversation
A connection between two transaction programs that allows them to exchange logical records containing a two-byte prefix that specifies the length of the record. LUs 1, 2, and 3 do not use the two-byte prefix. However, LU 1, 2, and 3 conversations must be basic conversations. This conversation type is used by service transaction programs as well as LU 1, 2, and 3 application transaction programs.

mapped conversation
A connection between two transaction programs that allows them to exchange data records of any length and in any format specified by the transaction programs. This conversation type is used for LU 6.2 conversations only and is used primarily for application transaction programs.

The following library subroutines can be used with any conversation type. Some of the requests that the **snactl** subroutine sends can only be used with a basic conversation. See the **snactl** subroutine for information about these restrictions.

- **snaclse**
- **snactl**
- **snadeal**
- **snalloc**
- **snaopen**
- **snaread**
- **snawrit**.

When using these subroutines, the general order of events is as follows:

1. Use the **snaopen** subroutine to initialize the connection to a remote node.
2. Use the **snalloc** subroutine to create a conversation between your transaction program and a transaction program at the remote node.
3. Begin data exchange with the remote program using:
   - the **snaread** subroutine to get data from the remote program
   - the **snawrit** subroutine to send data to the remote program
   - the **snactl** subroutine to monitor and control the conversation.

4. When the operation is complete, use the **snadeal** subroutine to dissolve the conversation between the two transaction programs.
5. Use the **snaclse** subroutine to end the connection.

AIX SNA Services/6000 Library Subroutines for Network Management (System Services Control Point Subroutines)

The second category of subroutines in the runtime library is network management subroutines. These subroutines send and receive network management information in the form of Network Management Vector Transport (NMVT) data, transmitting the data between AIX SNA Services/6000 in an AIX node and the System Services Control Point (SSCP) in the host through an SSCP-PU session.

Network management subroutines provide the following functions:

nm_open Associates the program with the SSCP-PU session.
nm_send Sends NMVT data to the SSCP-PU session.
nm_receive Receives NMVT data from the SSCP-PU session.
nm_close Releases the SSCP-PU session.
nm_status Obtains the status of the SSCP-PU session.

To send NMVT data, you need a Network Management Program that is defined as a server program and that includes the subroutines just described. The subroutines are used in the following sequence:

1. Associate program with the session, using the **nm_open** subroutine.
2. Format the NMVT data.
3. Send the NMVT data to the host, using the **nm_send** subroutine. The data is sent as a request.
4. Receive and check the response from the host, using the **nm_receive** subroutine.
5. Free the SSCP session, using the **nm_close** subroutine.

AIX SNA Services/6000 Special Files

luxsna.h Include File

Purpose

Defines constants and structures used by AIX SNA Services/6000 subroutines.

Path Name

/usr/include/luxsna.h

Description

The **luxsna.h** header file provides definitions of constants and structures that are used by AIX SNA Services/6000 subroutines.

Structures

The header file defines the following structures:

- **allo_str**
- **alloc_listen**
- **attr_str**
- **confirm_str**
- **cp_str**
- **deal_str**
- **erro_str**
- **ext_io_str**
- **flush_str**
- **fmh_str**
- **get_parms**
- **gstat_str**
- **pip_str**
- **prep_str**
- **read_out**
- **stat_str**
- **write_out**

allo_str Structure

This structure provides additional parameters for the **snalloc** subroutine and the **ioctl**(ALLOCATE) subroutine. Refer to the **snalloc** or **ioctl** subroutine in *Calls and Subroutines Reference* for a description of these functions. The structure appears as follows:

```
struct allo_str
{
    char        mode_name[9];
    char        tpn[65];
    unsigned    priority              :  2;
    unsigned    type                  :  2;
    unsigned    return_control        :  2;
    unsigned    sync_level            :  2;
    unsigned    recov_level           :  2;
    unsigned    pip                   :  1;
    unsigned    sess_type             :  1;
    unsigned    svc_tpn_flag          :  1;
    unsigned    rsvd2                 :  3;
    int         sense_code;
    long        rid;          /* rid for a previous conversation */
    struct      pip_str  *pip_ptr;
    int         conv_group_id;
} ;
```

Note: When parameters in this structure are specified, the remote TPN profile parameters are overriden. If no parameters are specified, the remote TPN profile parameters are used as the default.

The **allo_str** structure parameters have the following meaning:

`mode_name`
: Specifies the mode name for the conversation. The mode name designates the network properties for the session to be allocated, such as the class of service to be used.

  A special mode name, SNASVCMG, specifies the mode name used for control operator subroutines. Any program using this mode name must have control operator privileges based on the operating-system group ID. This mode name is not used for LU 1, 2, or 3.

`tpn`
: Specifies the name of the remote transaction program with which to establish the conversation. The TPN must be coded in EBCDIC. The AIX SNA Services/6000 subroutine interface converts ASCII to EBCDIC if the `svc_tpn_flag` field is set to zero. If you use the AIX SNA Services/6000 subroutine interface, the `svc_tpn_flag` field is ignored, and you must ensure that the conversion is done. Refer to EBCDIC to ASCII Translation for US English (TEXT) in *Communication Concepts and Procedures* for assistance in converting ASCII to EBCDIC.

`priority`
: Specifies the priority option which selects a mode profile to be used to establish an appropriate session for the conversation. Priority options are:

  B'00'
  : Use the first mode profile listed in the mode list profile for the connection. The mode list profile name is specified in the connection profile.

  B'01'
  : Use the second mode profile listed in the mode list profile for the connection. The mode list profile name is specified in the connection profile.

B'10' Use the third mode profile listed in the mode list profile for the connection. The mode list profile name is specified in the connection profile.

B'11' Use the fourth mode profile listed in the mode list profile for the connection. The mode list profile name is specified in the connection profile.

`type` Specifies the type of conversation to be allocated:

DEF_CONV (B'00') Use the value defined in the remote transaction profile.

BASIC_CONV (B'01')
Allocate a basic conversation.

MAPPED_CONV (B'10')
Allocate a mapped conversation. Do not use this value for LUs 1, 2, or 3.

RECON_CONV (B'11')
Reconnect a conversation between the local and remote transaction program. Do not use this value with LU 1, 2, or 3 conversations.

`return_control`
Specifies the type of conversation the application is requesting. Options are:

WHEN_SESSION_ALLOC (B'00')
Return control to application when either a Contention Winner or Contention Loser session is allocated. The type assigned is the first available.

WHEN_CONWINNER_ALLOCATED (B'01')
Return control to application when a Contention Winner session is allocated.

WHEN_CGID_ALLOCATED (B'10')
Return control to application when a session with the specified conversation group id is allocated.

RESERVED (B'11') Reserved.

`sync_level` Specifies the synchronization level to be used by the program for this conversation:

DEFAULT (B'00') Use the value defined in the remote transaction profile.

SYNC_NONE (B'01')
The program does not perform confirmation.

SYNC_CONF (B'10')
The program performs confirmation processing.

`recov_level` Specifies the recovery level that the local program uses for this conversation. Do not use this parameter with LU 1, 2, or 3.

RECOV_DEF (B'00')
Use the value defined in the remote transaction profile.

RECOV_NONE (B'01')
: The program does not support program reconnect or sync point restart.

RECOV_RECON (B'10')
: The program supports program reconnect, but not sync point restart.

`PIP`
: Specifies whether program initialization data is provided for the remote transaction program. Do not use this parameter with LU 1, 2, or 3 conversations.

PIP_NO (B'0')	Program initialization data is not provided.
PIP_YES (B'1')	Program initialization data is provided.

`sess_type`
: Specifies the type of session to be allocated. Use for LU 1, 2, or 3 sessions only. This field is not used for mapped conversations.

B'0'	LU-LU session
B'1'	SSCP-LU session

`svc_tpn_flag`
: Indicates whether the `tpn` parameter defines a service transaction program name specified in hex:

B'0'
: Indicates that it is not a service transaction program name. The AIX SNA Services/6000 subroutine interface translates the TPN name into EBCDIC code.

B'1'
: Indicates that it is a service transaction program name. AIX SNA Services/6000 does not translate the TPN name into EBCDIC code. Refer to EBCDIC to ASCII Translation for US English (TEXT) in *Communication Concepts and Procedures* for assistance in converting ASCII to EBCDIC.

`rsvd2`
: This field is not used.

`rid`
: Specifies a resource ID that is returned from an **ioctl**(ALLOCATE) or a **snalloc** subroutine. This field is also used to supply the `rid` necessary in order to reconnect to an old conversation when the type parameter of the **allo_str** structure is specified as *reconnect* (B'11'). For the remote attached ALLOCATE, the only parameter required in the **allo_str** structure is `rid`, which is passed into the application program by `argv [3]`.

`pip_ptr`
: When the PIP parameter indicates that program initialization data for the remote program is being supplied, this pointer points to the structure that contains that initialization data. Refer to the **pip_str** structure on page 8–24 for **PIP** data structure.

`conv_group_id`
: The conversation group ID that identifies a specific session to be allocated.

alloc_listen Structure

This structure provides additional parameters for the **ioctl**(ALLOCATE_LISTEN) subroutine. Refer to the **ioctl** subroutine in *Calls and Subroutines Reference* for a description of this function. The structure appears as follows:

```
struct alloc_listen
{
        int                     tpn_mask
        char                    tpn_profile_name[ MAX_PROF_LEN ];
        unsigned short  num_tpn;
        char                    tpn_list[ MAX_NUM_TPN ][MAX_TPN_LEN
];
} ;
```

The **alloc_listen** structure parameters have the following meaning:

`tpn_profile_name`
: Specifies the name of a Transaction Program Profile against which incoming attaches are checked. This profile serves as conversation characteristics template for the Transaction Programs listed in the `tpn_list` field. The name can be up to 15 bytes long and *must* be null-terminated (total of 16 bytes maximum). This profile name must be defined in the TPN List Profile for the specified connection.

`num_tpn`
: The number of TPNs in the `tpn_list` array described below.

`tpn_list`
: An array of Transaction Program Names to register as being "listened" for. A maximum of 31 names can be registered per call. Each name can be a maximum of 64 bytes and *must* be null-terminated (total of 65 bytes per name). These TPNs should not be defined in the TPN List Profile for the specified connection. Their conversation characteristics are defined in the single TPN profile specified in the `tpn_profile_name` field. These names are not converted from ASCII to EBCDIC.

`tpn_mask`
: A mask that indicates which of the specified TPNs are registered. The least significant bit (bit 0) corresponds to the first TPN (`tpn_list[0]`), and so on. The values of the bits are:

0 TPN is not registered.
1 TPN is registered.

If no TPN was registered, the result is (–1) or `0x7FFFFFFF`.

attr_str Structure

This structure receives output from the GET_ATTRIBUTE request for the **snactl** and **ioctl** subroutines. Refer to the **snactl** or **ioctl** subroutine in *Calls and Subroutines Reference* for a description of the these functions. The structure appears as follows:

```
struct attr_str
{
    long rid;
    char own_fully_qualified_lu_name[18];
    char ptner_fully_qualified_lu_name[18];
    char conversation_correlator[18];
    char modename[9];
    unsigned conn_status          :  1;
    unsigned sync_level           :  2;
    unsigned recovery_level       :  2;
    unsigned rsvd                 :  3;
    long conv_group_id
} ;
```

The **attr_str** structure parameters have the following meaning:

`rid`
: Specifies the variable that contains the resource ID of the GET_ATTRIBUTE conversation. This is the resource ID returned by the **snalloc** or **ioctl**(ALLOCATE) subroutine.

`own_fully_qualified_lu_name`
: The variable where the request returns the fully qualified name of the LU at which the local transaction program is located.

`ptner_fully_qualified_lu_name`
: The variable where the request returns the fully qualified name of the LU at which the remote transaction program is located.

`conversation_correlator`
: The variable where the request returns the local program's conversation correlator, an identifier that ties the current instance of the local program to the conversation.

`modename`
: The variable where the request returns the mode name for the session on which the conversation is allocated.

`conn_status`
: A variable where the request returns a value specifying the status of the connection. Valid values are:

B'0'	Connected
B'1'	Stopped.

`sync_level`
: A variable where the request returns a value specifying the level of synchronization processing being used for the conversation. Valid values are:

B'00'	No synchronization processing
B'01'	Confirm synchronization processing is being used.

`recovery_level`
: A variable where the request returns a value specifying the level of recovery being used for the conversation. Valid values are:

B'00'	No recovery level
B'01'	Reconnect.

rsvd This field is not used.

conv_group_id
A variable where the request returns the conversation group ID that identifies a specific allocated session.

confirm_str Structure

This structure receives output from the CONFIRM request for the **snactl** and **ioctl** subroutines. Refer to the **snactl** or **ioctl** subroutine in *Calls and Subroutines Reference* for a description of the these functions. The structure appears as follows:

```
struct confirm_str
{
   long rid;
   int sense_code;
} ;
```

The **confirm_str** structure parameters have the following meanings:

rid Specifies the variable that contains the resource ID of the conversation to perform the confirm function. This is the resource ID returned by the **snalloc** or **ioctl**(ALLOCATE) subroutine.

sense_code Specifies a variable that contains the sense code returned from AIX SNA Services/6000. This parameter is used for LUs 1, 2, and 3 only.

cp_str Structure

This structure provides additional parameters that describe the cp capabilities of the adjacent node. Refer to CP-Status in the **snactl** or **ioctl** subroutine in *Calls and Subroutines Reference* for a description of the these functions. This structure appears as follows:

```
struct cp_str
{
    long rid;
    char adj_cp_name[18];
    int conv_group_id;
    int sess_type;
    struct adj_cp_cap adj_cp_cap;
} ;
```

The **cp_str** structure parameters have the following meaning:

rid The variable that contains the resource ID returned from the allocate function.

adj_cp_name The variable where the request returns the adjacent control point (CP) name.

conv_group_id
The variable where the request returns the conversation group ID that identifies a specific allocated session.

sess_type The variable where the request returns the session type:

0 No session allocated
1 Contention Loser session
2 Contention Winner session.

```
struct adj_cp_cap
    {
    unsigned  locate_gds          : 1;
    unsigned  directory_svc       : 1;
    unsigned  resource_reg        : 1;
    unsigned  char_reg            : 1;
    unsigned  topology_update     : 1;
    unsigned  cp_msu              : 1;
    unsigned  unsol_cp_msu        : 1;
    unsigned  parallel_cp         : 1;
    int flow_reduction_seq_num;
    } ;
```

The **adj_cp_cap** structure parameters have the following meaning:

`locate_gds`
: Indicates whether the remote cp accepts LOCATE/CDINIT requests for resources that cp controls. The value returned is either TRUE (1) or FALSE (0).

`directory_svc`
: Indicates whether the remote cp forwards LOCATE/CDINIT search requests for resources to other cps. The value returned is either TRUE (1) or FALSE (0).

`resource_reg`
: Indicates whether the remote cp accepts register requests. The value returned is either TRUE (1) or FALSE (0).

`char_reg`
: Indicates whether the remote cp accepts register requests that include resource characteristics. The value returned is either TRUE (1) or FALSE (0).

`topology_update`
: Indicates whether the remote cp accepts topology database updates on the current session. The value returned is either TRUE (1) or FALSE (0).

`cp_msu`
: Indicates whether the remote cp accepts requests for management services data in a CP-MSU and replies to requests in a CP-MSU. The value returned is either TRUE (1) or FALSE (0).

`unsol_cp_msu`
: Indicates whether the remote cp accepts unsolicited requests for management services data in a CP-MSU and replies to requests in a CP-MSU. The value returned is either TRUE (1) or FALSE (0).

`parallel_cp`
: Indicates whether the remote cp supports and activates parallel CP-CP sessions. The value returned is either TRUE (1) or FALSE (0).

`flow_reduction_seq_num`
: The value the remote node last used to reduce the number of database updates sent when two nodes were reconnected.

deal_str Structure

This structure provides additional parameters for the **snadeal** and **ioctl**(DEALLOCATE) subroutines. Refer to the **snadeal** or **ioctl** subroutine in *Calls and Subroutines Reference* for a description of the these functions. This structure is as follows:

```
struct deal_str
{
    long  rid;
    unsigned  type          :  3;
    unsigned  deal_flag :  1;
    unsigned  rsvd          : 12;
    int  sense_code;
} ;
```

The **deal_str** structure parameters have the following meaning:

`rid` Specifies the variable that contains the resource ID of the conversation to be deallocated. This is the resource ID returned by the **snalloc** or **ioctl**(ALLOCATE) subroutine.

`type` Specifies the type of deallocation to be performed for this conversation. This parameter is optional. If you do not provide a value, the system uses a value of B'000'. Because this value cannot be used for LU 1, 2, or 3, you must specify a *type* value when using those logical unit specifications. Values for type are:

DEFAULT (B'000') Use the default value specified in the sync_level parameter of the **snalloc** or **ioctl**(ALLOCATE) subroutine that established this conversation. Do not use this value for LU 1, 2, or 3.

DEAL_CONFIRM (B'001')
Perform CONFIRM logic (see the **snactl**(CONFIRM) or **ioctl**(CONFIRM) subroutine). If that request is successful, deallocate the conversation normally; otherwise, the conversation remains allocated. Do not use this value for LU 2 or 3.

DEAL_ABEND (B'010')
Deallocate the conversation abnormally. Do not use this value for LU 1, 2, or 3.

DEAL_FLUSH (B'011')
Flush the send buffer and deallocate the conversation normally.

`deal_flag` Specifies whether the resource ID is discarded or retained when the conversation is deallocated. This parameter is optional. If you do not provide a value, the system uses a value of B'0'. Values for deal_flag are:

DISCARD (B'0') Specifies that the resource ID be discarded. The local transaction program will not reconnect to the conversation.

RETAIN (B'1') Specifies that the resource ID be retained. The local transaction program plans to reconnect to the conversation. Do not use this value with LU 1, 2, or 3.

rsvd — This field is not used.

sense_code — Specifies a variable that contains the sense code returned from AIX SNA Services/6000. This parameter is used for LUs 1, 2, and 3 only.

erro_str Structure

This structure provides additional parameters for the SEND_ERROR request for the **snactl** and **ioctl** subroutines. Refer to the **snactl** or **ioctl** subroutine in *Calls and Subroutines Reference* for a description of the these functions. The structure appears as follows:

```
struct erro_str
{
    long rid;
    int  sense_code;
    unsigned type  :  1;
    unsigned rsvd  : 15;
} ;
```

The `rid` parameter is the only parameter used for a mapped conversation.

rid — Specifies the variable that contains the resource ID of the conversation to perform the send_error function. This is the resource ID returned by the **snalloc** or **ioctl**(ALLOCATE) subroutine.

The additional parameters for a basic conversation have the following meanings:

type — Specifies the level of error being reported, as follows:

- B'0' — An application program produced the error being reported.
- B'1' — LU services Transaction Program produced the error being reported.

This parameter is optional. If not specified, SNA services provides a value of B'0'. This parameter is used for LU 6.2 Basic conversations only.

sense_code — Specifies a variable that contains the sense code to be reported to the remote session. This parameter is used for LUs 1, 2, and 3 only.

rsvd — This field is not used.

ext_io_str Structure

This structure provides additional input and output parameters for the **readx** and **writex** subroutines. Neither subroutine uses all parameters in the structure. Refer to the **readx** and **writex** subroutines in *Calls and Subroutines Reference* for a description of the functions provided and the fields used.

The structure appears as follows:

```
struct ext_io_str
{
    struct input
    {
        unsigned priority                :  2;
        unsigned tpn_option              :  2;
        unsigned confirm                 :  1;
        unsigned deallocate              :  1;
        unsigned deallo_type             :  3;
        unsigned deallo_flag             :  1;
        unsigned allocate                :  1;
        unsigned fill                    :  1;
        unsigned mc_gds                  :  1;
        unsigned sess_type               :  1;
        unsigned flush_flag              :  2;
    } input;
    struct output
    {
        unsigned rq_to_snd_rcvd          :  1;
        unsigned what_data_rcvd          :  3;
        unsigned what_control_rcvd       :  5;
        unsigned usr_trunc               :  1;
        unsigned rsvd3                   :  6;
        unsigned gdsid                   : 16;
        int sense_code;
    } output;
    long rid;
    int usrhdr_len;
} ;
```

The **ext_io_str** structure parameters have the following meanings:

Input Parameters

These parameters are sent to SNA.

`priority` Specifies the priority option which selects a mode profile to be used to establish an appropriate session for the conversation. The priority option should be used with the **writex** subroutine only and should not be used with LU 1, 2, or 3. Priority options are:

B'00' Use the first mode profile listed in the mode list profile for the connection. The mode list profile name is specified in the connection profile.

B'01' Use the second mode profile listed in the mode list profile for the connection. The mode list profile name is specified in the connection profile.

B'10' Use the third mode profile listed in the mode list profile for the connection. The mode list profile name is specified in the connection profile.

B'11' Use the fourth mode profile listed in the mode list profile for the connection. The mode list profile name is specified in the connection profile.

`tpn_option` Specifies the remote transaction program name (RTPN) option which selects a remote RTPN profile to be used to establish an appropriate session for the conversation. The following RTPN options are used by the **writex** subroutine only:

B'00' Use the first RTPN profile listed in the RTPN list profile for the connection. The RTPN list profile name is specified in the connection profile.

B'01' Use the second RTPN profile listed in the RTPN list profile for the connection. The RTPN list profile name is specified in the connection profile.

B'10' Use the third RTPN profile listed in the RTPN list profile for the connection. The RTPN list profile name is specified in the connection profile.

B'11' Use the fourth RTPN profile listed in the RTPN list profile for the connection. The RTPN list profile name is specified in the connection profile.

`confirm` This parameter is used by the **writex** subroutine only and designates whether to flush the send buffer and wait for confirmation of receipt of the data from the remote application program.

B'0' Do not issue a CONFIRM.

B'1' Issue a CONFIRM.

`deallocate` This parameter is used by both the **writex** and **readx** subroutines and designates whether to deallocate the conversation after transmitting the data associated with this subroutine:

B'0' Do not deallocate the conversation.

B'1' Deallocate the conversation. If used with an SSCP-LU application program, it could also terminate the associated LU-LU session.

`deallo_type` This parameter specifies the type of deallocation to perform when a deallocation is performed along with the subroutine:

B'000' Deallocate the conversation as specified in the sync_level parameter used in the **ioctl**(ALLOCATE) subroutine request that established this conversation. Used by the **writex** subroutine only.

B'001' Issue a CONFIRM request. If that request is successful, deallocate the conversation normally; otherwise, the conversation remains allocated. Used by the **writex** subroutine only. Do not use this value for LUs 1, 2, or 3.

B'010' Deallocate the conversation abnormally. Used by the **writex** and **readx** subroutines only.

B'011' Flush the send buffer when the conversation is in the send state and deallocate the conversation normally. Used by the **writex** subroutine only.

`deallo_flag` Specifies whether the resource ID is discarded or retained when the conversation is deallocated. Used by the **writex** and **readx** subroutines only. Values for `deallo_flag` are:

DISCARD (B'0') Specifies that the resource ID be discarded. The local transaction program will not reconnect to the conversation.

RETAIN (B'1') Specifies that the resource ID be retained. The local transaction program plans to reconnect to the conversation. Do not use this value with LUs 1, 2, or 3.

`allocate` This parameter specifies whether to allocate a conversation along with the subroutine. Used by the **writex** and **readx** subroutines only:

B'0' Do not allocate a conversation. The `rid` field must be supplied.

B'1' Allocate a conversation. If the `rid` parameter is 0, allocate a new conversation. If the `rid` parameter is not 0, reconnect a previous conversation identified by the value of `rid`.

The `allocate` parameter can be used with the `deallocate` parameter (but not for LU 1, 2, or 3). If deallocate is also set, the **readx** and **writex** subroutines perform the following actions:

1. Allocates a conversation as described above.
2. Transfers the data associated with the subroutine.
3. Deallocates the conversation.

`fill` Specifies whether the program receives data without regard to the logical record format of the data. This parameter is optional and is used by the **readx** subroutine only. If you do not specify one of the two following values, the program uses a value of B'0'. Always use a value of B'0' for mapped conversations.

BUFFER (B'0') Specifies that the program receives data without regard to the logical record format of the data.

LL (B'1') Specifies that the program receives one complete logical record, or a logical record that has been truncated to the length specified in the *length* parameter of this subroutine. Do not use with LU 1, 2, or 3.

`mc_gds` Reserved for use by mapped conversation RTS.

`sess_type` Specifies the type of session to be allocated if the allocate field indicates that a session should be allocated. Used for LU 1, 2, and 3. Valid values are:

B'1' SSCP-LU session
B'0' LU-LU session.

`flush_flag` This parameter indicates whether to perform a **flush** (of the LU's send buffers) request in addition to the requested **writex** operation:

I_O_NO_FLUSH (B'00')
Do not perform the **flush**.

I_O_FLUSH_NOT_EC (B'01')
: Perform the **flush**, but do not indicate end of chain.

I_O_FLUSH_EC (B'10')
: Perform the **flush**, indicating end of chain. This function is used by LUs 1, 2, and 3 only.

Output Parameters

These parameters are set by SNA.

`rq_to_snd_rcvd`

Indicates whether a request to send has been received. The `rq_to_snd_rcvd` parameter specifies the variable used by the **writex** and **readx** subroutines only:

B'0'
: A request to send has not been received from the remote transaction program.

B'1'
: A request to send has been received from the remote transaction program.

`what_data_rcvd`

Specifies the variable that gets set to indicate what type of data the program received. Used by the **readx** subroutine only:

DATA (B'000')
: Indicates that data has been received by the program. Occurs only when the *fill* parameter for this call is **buffer**. Not used for LU 1, 2, or 3.

DATA_COMP (B'001')
: LU 6.2: Indicates that a complete logical record, or the last remaining portion of a logical record, has been received by the program. Occurs only when the *fill* parameter for this call is **ll**.

: LUs 1, 2, 3: Indicates that a complete chain element was received.

DATA_INCOMP (B'010')
: LU 6.2: Indicates that less than a complete logical record has been received by the program. Occurs only when the *fill* parameter for this call is **ll**.

: LUs 1, 2, 3: Indicates that a complete chain element was not received.

LL_TRUNCATED (B'011')
: Indicates that the 2-byte **ll** field of a logical record was truncated after the first byte and that the LU has discarded the **ll** field. The program does not receive the ll field. Not used for LU 1, 2, or 3.

FMH_COMPLETE (B'100')
: Indicates that the data received was FM header data for an LU 1 session and that the complete header has been received.

FMH_INCOMPLETE (B'101')
: Indicates that the data received was FM header data for an LU 1 session and that the complete header has not been received.

B'110'
: Not used.

B'111'
: Not used.

`what_control_rcvd`

Specifies the variable that is set to indicate the type of control that the program received. Used by the **readx** subroutine only:

B'X0000'
: No control information was received.

SEND (B'X0001')
: Indicates that the remote program has entered the receive state, placing the local program in the send state.

CONFIRM (B'X0010')
: Indicates that the remote program issued a CONFIRM request. The local program must respond with an **ioctl** subroutine, using either a CONFIRMED request or a SEND_ERROR request.

CONFIRM_SEND (B'X0011')
: Indicates that the remote program used an **ioctl** or **snactl** subroutine to issue a PREPARE_TO_RECEIVE request, and that the type parameter was set to B'10' (confirm).

CONFIRM_DEALLOCATE (B'X0100')
: Indicates that the remote program used an **ioctl** or **snactl** subroutine to issue a DEALLOCATE request with the type parameter set to B'001' (confirm). Not used for SSCP-LU sessions for LU 1, 2, or 3.

NORMAL_DEALLOCATE (B'X0101')
: Indicates that the remote program issued a DEALLOCATE request with the type parameter set to B'011' (flush). Not used for SSCP-LU sessions for LU 1, 2, or 3.

CONFIRM_DEALLOCATE_RETAIN (B'X0110')
: Indicates that the remote program issued a DEALLOCATE request with the type parameter set to B'001' (confirm) and the deal_flag parameter set to retain. Not used for LU 1, 2, or 3.

NORMAL_DEALLOCATE_RETAIN (B'X0111')
: Indicates that the remote program issued a DEALLOCATE request with the type parameter set to B'011' (flush) and the deal_flag parameter set to retain. Not used for LU 1, 2, or 3.

Note: The following parameters are used by LU 1, 2, or 3 only:

FLUSH RECEIVED (B'X1000')
: Specifies end chain, RQE, not change direction.

NOT END OF DATA (B'X1001')
: Specifies end chain was not received from the host.

BEGIN CHAIN (B'1XXXX')
: Specifies begin chain indicator was received from the host. This may be set in addition to other returned flags.

NOT BEGIN CHAIN (B'0XXXX')
: Specifies begin chain was not received in this data buffer.

Note: 'X' means that the X bit positions are meaningless for this function.

`usr_trunc`
: Indicates that the length of the user header field was not large enough for the received header data. You can get no information from the user header field when the user header has been truncated.

`rsvd3`
: This field is not used.

`gdsid`
: This field is used for mapped conversation runtime service (RTS) routines only.

`sense_code`
: Specifies the variable that is set to the value of the sense code for negative responses received. Used for LUs 1, 2, and 3 only.

`rid`
: This parameter specifies the resource ID returned by the **snalloc** or **ioctl**(ALLOCATE) subroutine for this connection. This parameter has the following effects:

- For the **writex** subroutine with the allocate bit on and `rid` equal to 0, the system allocates a new conversation and returns the resource ID in `rid`.
- For the **writex** subroutine with the allocate bit on and `rid` not equal to 0, the system reconnects an old conversation identified by the value in `rid`.
- For the **readx** subroutine with the allocate bit on and `rid` not equal to 0, it indicates a remotely attached allocation.

`usrhdr_len`
: This parameter specifies the number of bytes of header data to be sent or that was received with the data. The header data is provided in the **usrhdr** field (see "User Header Field" for the **writex** subroutine or "User Header Field" for the **readx** subroutine.).

flush_str Structure

This structure provides additional parameters for the FLUSH request for the **snactl** and **ioctl** subroutine. Refer to the **snactl** or **ioctl** subroutine in *Calls and Subroutines Reference* for a description of the these functions. The structure appears as follows:

```
struct flush_str
{
      long rid;
      unsigned end_chain      :  1;
      unsigned rsvd           : 15;
      int sense_code;
} ;
```

The **flush_str** structure parameters have the following meanings:

`rid`
: This parameter specifies the resource ID returned by the **ioctl**(ALLOCATE) or the **snalloc** subroutine for this connection.

`end_chain`
: Specifies whether or not to send the buffer with the *end chain* indication. The program specifies this parameter as a 1 to complete a chain. To flush the send buffer without completing the chain, the program specifies this parameter as a 0. Valid values are:

  0 Send buffer without *end chain*.
  1 Send buffer with *end chain*.

`rsvd`
: This field is not used.

`sense_code`
: Specifies a field that receives indications of errors that occurred on previously sent data. Used for LU 1, 2, and 3 only.

fmh_str Structure

This structure used for LU 1, 2, and 3 provides additional parameters for the SEND_FMH request for the **snactl** and **ioctl** subroutines. Refer to the **snactl** or **ioctl** subroutine in *Calls and Subroutines Reference* for a description of the these functions. The structure appears as follows:

```
struct fmh_str
{
      long rid;
      short fmh_len;
      unsigned type           :  2;
      unsigned rsvd           : 14;
      char *fmh_addr;
      int sense_code;
} ;
```

The **fmh_str** structure parameters have the following meanings:

`rid`
: Specifies the variable that contains the resource ID of the conversation to perform the SEND_FMH request. This is the resource ID returned by the **snalloc** or **ioctl**(ALLOCATE) subroutine.

`fmh_len`
: Specifies the length (in bytes) of the FM header to be sent.

`type`
: Specifies the type of request to be performed for this conversation. Values for type are:

  B'00' Flush without end chain.

  B'01' Flush the FMH with end chain.

  B'10' Execute a CONFIRM function (see the **snactl**(CONFIRM) subroutine).

  B'11' Do not flush the FMH.

`rsvd`
: This field is not used.

`fmh_addr`
: Specifies a pointer to the address of the FM header to be sent.

`sense_code`
: Specifies a variable that contains the sense code returned from the AIX SNA Services/6000.

get_parms Structure

Returns data associated with the **get_parameters** structure of the **ioctl** subroutine.

```
struct get_parms
{
      int gparms_size;
      char gparms_data [MAX_GETPARMS_DATA];
} ;
```

The **get_parms** structure parameters have the following meanings:

`gparms_size` Specifies the length (in bytes) of the parameter data returned in the `gparms_data` field.

`gparms_data` Contains the following data:

TPN Name Specifies the transaction program name, which has a maximum length of 64 bytes plus a terminating blank.

Connection Name Specifies the connection name, which has a maximum length of 15 bytes plus a terminating blank.

RID Specifies the resource ID, which has a maximum length of 6 bytes plus a terminating blank.

PIP Specifies the program initialization parameters, which have a maximum of sixteen fields, each having a maximum of 64 bytes plus a terminating blank.

gstat_str Structure

This structure provides current link and session information in response to a GET_STATUS request with either an **ioctl** or **snactl** subroutine. Refer to the **snactl** or **ioctl** subroutine in *Calls and Subroutines Reference* for a description of the these functions. This structure is defined for use by LUs 1, 2, and 3 only. The structure appears as follows:

```
struct gstat_str
{
      int sscp_sense_code;
      int status;
      unsigned  rtn_image          :  1;
      unsigned  rsrv               : 15;
      short  image_len;
      char  *image;
      int lu_sense_code;
} ;
```

The **gstat_str** structure parameters have the following meaning:

`status` Specifies the current status of the physical and logical link and the SSCP-LU and LU-LU sessions. Several values may be set at once (bit settings). Valid values shown in parentheses are expressed in hexadecimal notation below:

LINK_ACTIVE (0X1) The link is active.

LINK_INACTIVE (0X2) The link is not active.

LINK_TIMEOUT (0X4) A link time out has occurred.

SSCP_LU_ACTIVE (0X8) An SSC-LU session is active on the link.

SSCP_LU_INACTIVE (0X10)
An SSCP-LU session is allocated to the link, but is not active.

LU_LU_ACTIVE (0X20) An LU-LU session is allocated to the link and is currently active.

LU_LU_INACTIVE (0X40) An LU-LU session is allocated to the link, but is not active.

LU_LU_RESET (0X80) The LU-LU session allocated to the link has been reset.

HOST_BID (0X100) A BID, request to begin a bracket, was received from the host.

LU_LU_SHUTDOWN (0X200)
The LU-LU session has been shut down.

LU_LU_CONFIRM_RCVD (0X400)
A CONFIRM request is received from the host on the LU-LU session.

SSCP_LU_CONFIRM_RCVD (0X800)
A CONFIRM request is received from the host on the SSCP-LU session.

RSP_TO_RTS_RCVD (0X1000)
A response is received from the host for a request to send, sent on the LU-LU session.

RQTS_RCVD (0X2000) A request to send is received from the host.

RSP_TO_CANCEL_RCVD (0X4000)
A response is received from the host for a CANCEL request previously sent on the LU-LU session.

`rtn_image` When set, this field indicates that the BIND image associated with the session should be returned in the buffer pointed to by image_ptr.

`image_len` This field contains one of the following values if `rtn_image` is set:

- When the GET_STATUS is issued, this field contains the maximum amount of BIND image data (in bytes) that can be returned by the request.
- When the request is complete, this field contains the actual amount of BIND image data (in bytes) that was returned.

`image` Specifies a pointer (up to 26 bytes or the image length) to the buffer area in which the BIND image data is to be stored.

sscp_sense_code
: This parameter is set to 0 if the response is positive and to the sense code received if negative.

lu_sense_code
: This parameter is set to 0 if the response is positive and to the sense code received if negative.

pip_str Structure

This structure provides program initialization parameters (PIP) to be sent to a remote program by the **snalloc** or **ioctl**(ALLOCATE) subroutine. Refer to the **snalloc** or **ioctl** subroutine in *Calls and Subroutines Reference* for a description of the these functions. This structure is defined for use by LU 6.2 only. The structure appears as follows:

```
struct pip_str
{
      int          sub_num;
      char         sub_data [16][65].;
} ;
```

The **pip_str** structure parameters have the following meaning:

sub_num — A variable that specifies the number of PIP subfields in `sub_data`.

sub_data — The array of program initialization data for the remote program. There may be up to 16 entries, each containing up to 64 bytes of initialization data.

prep_str Structure

This structure provides additional parameters for the PREPARE_TO_RECEIVE request for the **snactl** and **ioctl** subroutines. Refer to the **snactl** or **ioctl** subroutine in *Calls and Subroutines Reference* for a description of the these functions. The structure appears as follows:

```
struct prep_str
{
      long rid;
      unsigned type      :  2;
      unsigned rsvd      : 14;
      int sense_code;
} ;
```

The **prep_str** structure parameters have the following meanings:

rid — This parameter specifies the resource ID returned by the **ioctl**(ALLOCATE) or **snactl**(ALLOCATE) subroutine for this connection.

type — Specifies the type of request to be performed for this conversation. Values for `type` are:

SYNCL_DEF (B'00') — Use the default value specified in the `sync_level` parameter of the **snalloc** subroutine that established this conversation.

SYNCL_NONE (B'01') — Flush the send buffer and enter the receive state.

SYNCL_CONFIRM (B'10') — Execute a confirm function (see the **snactl**(CONFIRM) subroutine). If that request is successful, enter the receive state.

SYNCL_FLUSH (B'11')
: Flush the send buffer and enter the receive state.

rsvd
: This field is not used.

sense_code
: Specifies a field that receives indications of errors that occurred on previously sent data. This parameter is used by LUs 1, 2, and 3 only.

read_out Structure

This structure receives output from the **snaread** subroutine. Refer to the **snaread** subroutine in *Calls and Subroutines Reference* for a description of that function. The structure appears as follows:

```
struct read_out
{
      long        rid;
      int         request_to_send_received;
      int         what_data_rcvd;
      int         what_control_rcvd;
      int         sense_code;
} ;
```

The **read_out** structure parameters have the following meanings:

rid
: Specifies the variable that contains the resource ID returned by the **snalloc** subroutine that allocated the resource to be read.

request_to_send_received
: Specifies the variable that gets set to indicate whether a request to send has been received:

TRUE(1)
: A request to send has been received from the remote transaction program.

FALSE (0)
: A request to send has not been received from the remote transaction program.

what_data_rcvd
: Specifies the variable that gets set to indicate what type of data the program received:

0
: Indicates that data has been received by the program. Occurs only when the *fill* parameter for this subroutine is buffer.

DATA_COMPLETE (1)
: Occurs only when the *fill* parameter for this subroutine is **ll**. It indicates that a complete logical record, or the last remaining portion of a logical record, has been received by the program.

DATA_INCOMPLETE (2)
: Occurs only when the *fill* parameter for this subroutine is **ll**. It indicates that less than a complete logical record has been received by the program.

3
: Indicates that the 2-byte ll field of a logical record was truncated after the first byte and that the LU has discarded the **ll** field. The program does not receive the **ll** field. Not used for LU 1, 2, or 3.

4 — Indicates that the data received was FM header data for an LU 1 session and that the complete FM header was received.

5 — Indicates that the data received was FM header data for an LU 1 session, but that the complete FM header was not received.

`what_control_rcvd`
Specifies the variable that is set to indicate the type of control that the program received:

0 — No control information received.

1 — Indicates that the remote program has entered the receive state, placing the local program in the send state.

2 — Indicates that the remote program used a snactl subroutine to issue a CONFIRM request. The local program must respond with a **snactl** subroutine using either a CONFIRMED request or a SEND_ERROR request.

3 — Indicates that the remote program used a **snactl** subroutine to issue a PREPARE_TO_RECEIVE request, and that the type parameter was set to B'10' (confirm).

4 — Indicates that the remote program used a **snadeal** subroutine with the type parameter set to B'001' (confirm). Not used for SSCP-LU sessions for LU 1, 2, or 3.

5 — Indicates that the remote program used a **snadeal** subroutine with the type parameter set to B'011' (flush). Not used for SSCP-LU sessions for LU 1, 2, or 3.

6 — Indicates that the remote program used a **snadeal** subroutine with the type parameter set to B'001' (confirm) and the deal_flag parameter set to retain. Not used for LU 1, 2, or 3.

7 — Indicates that the remote program used a **snadeal** subroutine with the type parameter set to B'011' (flush) and the deal_flag parameter set to retain. Not used for LU 1, 2, or 3.

`sense_code` Specifies the variable that is set to the value of the sense code for negative responses. Used for LUs 1, 2, and 3 only.

stat_str Structure

This structure provides additional parameters for the SEND_STATUS request for the **snactl** and **ioctl** subroutines. Refer to the **snactl** or **ioctl** subroutine in *Calls and Subroutines Reference* for a description of the these functions. The structure appears as follows:

```
struct stat_str
{
      long rid;
      unsigned type          : 4;
      unsigned id      : 8;
      int  sense_code;
} ;
```

The **stat_str** structure parameters have the following meanings:

`rid` Specifies the variable that contains the resource ID returned by the **snalloc** or **ioctl**(ALLOCATE) subroutine that performs the SEND_STATUS function.

`type` Specifies the status condition to be reported. Use one of the following values in parentheses as defined in the **luxsna.h** header file:

- POWER_ON (0) The device is on.
- POWER_OFF (1) The device is off.
- UNAVAILABLE (2) The device is not configured.
- PERMANENT_ERROR (3) The device has an error which cannot be corrected.
- PS_ALTERED (4) Presentation space altered.
- UNBIND_REQUESTED (5) A request shutdown (RSHUTD) command has been sent to the partner LU requesting an unbind (UNBIND) command to be sent to this secondary LU. This *type* does not use the *id* field. This *type* does not cause the received data to be rejected. The application program should continue to read the data until it receives SNA_NSES (session not active).
- ATTENDED (6) The device is attended by an operator (LU 1 only).
- UNATTENDED (7) The device is no longer attended by an operator (LU 1 only).

`ID` Specifies the ID of the device for which status is being reported.

`sense_code` Specifies a variable that contains the sense code returned from AIX SNA Services/6000.

write_out Structure

This structure provides additional parameters for the **snawrit** subroutine. Refer to the **snawrit** subroutine in *Calls and Subroutines Reference* for a description of these functions. The structure appears as follows:

```
struct write_out
{
      int request_to_send_received;
      int sense_code;
} ;
```

The **write_out** structure parameters have the following meanings:

`request_to_send_received`
Specifies the variable that gets set to indicate whether a request to send has been received:

TRUE(1) — A request to send has been received from the remote transaction program.

FALSE (0) — A request to send has not been received from the remote transaction program.

`sense_code` — Specifies the variable that is set to the value of the sense code for negative responses. Used for LUs 1, 2, and 3 only.

Constant Definitions

The **luxsna.h** file contains constant definitions that are used in the following areas of AIX SNA Services/6000:

- Status codes (see the **gstat_str** structure)
- Error codes
- Request codes for the **snactl** and **ioctl** subroutines.

Error Code Constants

This file defines the error return values that are exclusive to AIX SNA Services/6000. The AIX SNA Services/6000 subroutines set the **errno** global variable to one of the following values when an error occurs to indicate the nature or cause of the error.

Error Code Constants		**Page 1 of 3**
Name	**Code**	**Definition**
SNA_CTYPE	101	The specified conversation type does not match the indicated conversation.
SNA_NREC	103	Reconnect is not supported.
SNA_NSYC	104	Sync level is not supported.
SNA_ALFN	105	An allocation failure occurred. Do not try the operation again.
SNA_ALFR	106	An allocation failure occurred. Try the operation again.
SNA_LUNREC	107	Reconnect is not supported by the LU.
SNA_LUNSYC	108	Sync level is not supported by the LU.
SNA_RID	109	The resource ID was invalid.
SNA_STATE	110	The network management request was issued while the Program was not in an allowed state.
SNA_RFR	111	A resource failure occurred. Try the operation again.
SNA_RFN	112	A resource failure occurred. Do not try again.
SNA_PROTOCOL	113	An SNA protocol violation occurred.
SNA_NPIP	114	Remote program initialization parameter (PIP) data is not supported.
SNA_PNSYC	115	Sync level is not supported by the program.
SNA_PNREC	116	Reconnection is not supported by the program.

Error Code Constants		Page 2 of 3
Name	**Code**	**Definition**
SNA_NRREC	117	Could not reconnect to the transaction program. Do not try again.
SNA_PPURG	118	Program error purging.
SNA_PNTR	119	A program error occurred since no truncation is allowed.
SNA_PTR	120	Program error truncate.
SNA_PGMDEAL	121	A deallocation occurred due to the abnormal ending of the remote program.
SNA_BOUNDARY	122	The function was not requested on a logical record boundary.
SNA_NOMODE	123	Invalid mode name specified.
SNA_RREC	124	Cannot connect to the transaction program. Try the operation again.
SNA_NOCONN	125	The SNA connection has been stopped.
SNA_NRESTART	126	A recovery_level value of restart is not valid for this subroutine.
SNA_NOTPN	127	The specified transaction program name is not valid.
SNA_NRMDEAL	129	A normal deallocation terminated the conversation.
SNA_SVCDEAL	130	A deallocation occurred due to an abnormal ending of an system service (systems logic error).
SNA_TIMDEAL	131	Deallocate abend due to excessive time having elapsed.
SNA_WRGPIP	132	The remote program initialization data (PIP) specified was not correct.
SNA_INVACC	133	Access security information invalid.
SNA_SPURG	134	A SVC error occurred; purging.
SNA_SNTR	135	A service transaction program error occurred; no truncate.
SNA_STR	136	A service transaction program error occurred; truncate.
SNA_NDELAY	137	Delay allocation not supported.
SNA_SVCTYPE	138	An unsupported type was specified.
SNA_NFMH	139	The FM Header data gds variable is not supported by mapped conversation.
SNA_NMAPPING	140	The MAP name is not supported by mapped conversation.
SNA_MAP_NOT_FOUND	141	Map name not found.
SNA_MAPEXEC	142	Map execution failure.
SNA_GDSID	143	Invalid GDS identifier in data.
SNA_SHUT	144	A shutdown request was received.
SNA_NSES	145	The session is not established (LUs 1, 2, 3, and 6.2) or not active (LUs 1, 2, and 3).
SNA_PARMS	146	Input parameters not valid.
SNA_NTPN	147	Transaction Program cannot be started, no retry.

Error Code Constants		Page 3 of 3
Name	**Code**	**Definition**
SNA_NTPR	148	Transaction Program cannot be started due to lack of resources, retry.
SNA_RCANC	149	Received cancel for LU 1, 2, or 3.
SNA_SENSE	150	Sense code available for LU 1, 2, or 3 (exception request or negative response received).
SNA_NOTCP	151	This is not a CP connection.
SNA_FAIL	160	There was an SNA system failure.
SNA_NSACT	161	No session can be started as the session limit is set to 0.
SNA_NSLMT	162	No session can be activated as the number of sessions of the requested type has been exceeded.
SNA_INOP	164	Link INOP received.
SNA_HIER_RESET	165	Hierarchical reset received.
SNA_NO_LU	166	No LUs registered for generic SNA.
SNA_INUSE	170	The session between the system services control point and the physical unit is being used by another application.
SNA_NOTAVAIL	171	The requested session between the system services control point and the physical unit was not available.
SNA_UNDEF_SVR	172	The application server is not defined.
SNA_INVALID	173	The ID specified for the session between the system services control point and the physical unit (SSCP_ID) is not valid.
SNA_LENGTH	174	The length specified for the NMVT data is not valid.
SNA_ERP	175	The physical unit is not active. An error-recovery procedure (ERP) instructing you to activate the physical unit (ACTPU) was received.
SNA_INACT	176	The session between the system services control point and the physical unit is inactive.

Request Code Constants

This file defines the following constants and their codes for use in the *request* parameter of the **ioctl** or **snactl** subroutine.

Request Code Constants		Page 1 of 2
Name	**Code**	**Definition**
ALLOCATE	1	Allocates a conversation. Used only by the **ioctl** subroutine.
DEALLOCATE	2	Deallocates a conversation. Used only by the **ioctl** subroutine.
CONFIRM	3	Sends a request for confirmation of transmission to the remote transaction program.
CONFIRMED	4	Positive response to a CONFIRM request.

Request Code Constants		Page 2 of 2
Name	**Code**	**Definition**
FLUSH	5	Transmits everything in the send buffer to the remote transaction program. Used only for LU 1, 2, or 3.
PREPARE_TO_RECEIVE	6	Changes the conversation direction to allow the local transaction program to receive.
HIER_RESET_RSP	6	Hierarchical reset response for Generic SNA Applications.
REQUEST_TO_SEND	7	Request to change the conversation direction to allow the local transaction program to send.
INOP_RSP	7	Link inoperative response for Generic SNA Applications.
SEND_FMH	8	Sends the FM header to the remote LU. Used by the **snactl** subroutine for LU 1 on a basic conversation only. Used for LU 1, 2, or 3 only.
SEND_ERROR	9	Negative response to a CONFIRM request or incorrect data received.
GET_ATTRIBUTE	10	Gets information about the specified LU 6.2 conversation.
SEND_STATUS	11	Sends status information about the devices on the local session (LUs 1, 2, and 3, only) to the host program.
GET_STATUS	12	Gets information about the current link and session on a basic conversation only. Used only for LU 1, 2, or 3.
CP_STATUS	13	Requests the control point name, the session type, and the control point capabilities of the remote node.
ALLOCATE_LISTEN	14	Registers a list of transaction program names (TPNs) for which an application wishes to accept allocate requests. Used for LU 6.2 only.
GET_PARAMETERS	15	Retrieves the data associated with the receipt of an allocate request for a registered TPN on a particular connection. The GET_PARAMETERS argument is used in conjunction with the ALLOCATE_LISTEN argument. Used for LU 6.2 only.

Developing Special SNA Functions

Document Guide

Intended Readership

This document is intended for persons interested in developing special SNA functions not provided by AIX SNA Services/6000. Intended readers include both IBM programming personnel and third party programmers.

Reader Pre–Conditions

The reader should have a basic understanding of AIX system architecture, the AIX operating system and SNA concepts.

Reader Post–Conditions

After completing this document, the reader should:

- Have an understanding of the generic SNA device driver and its relationship to AIX SNA Services/6000.
- Be able to use the API provided by this document to write applications to perform generic SNA functions.

Recommended Reference Documentation

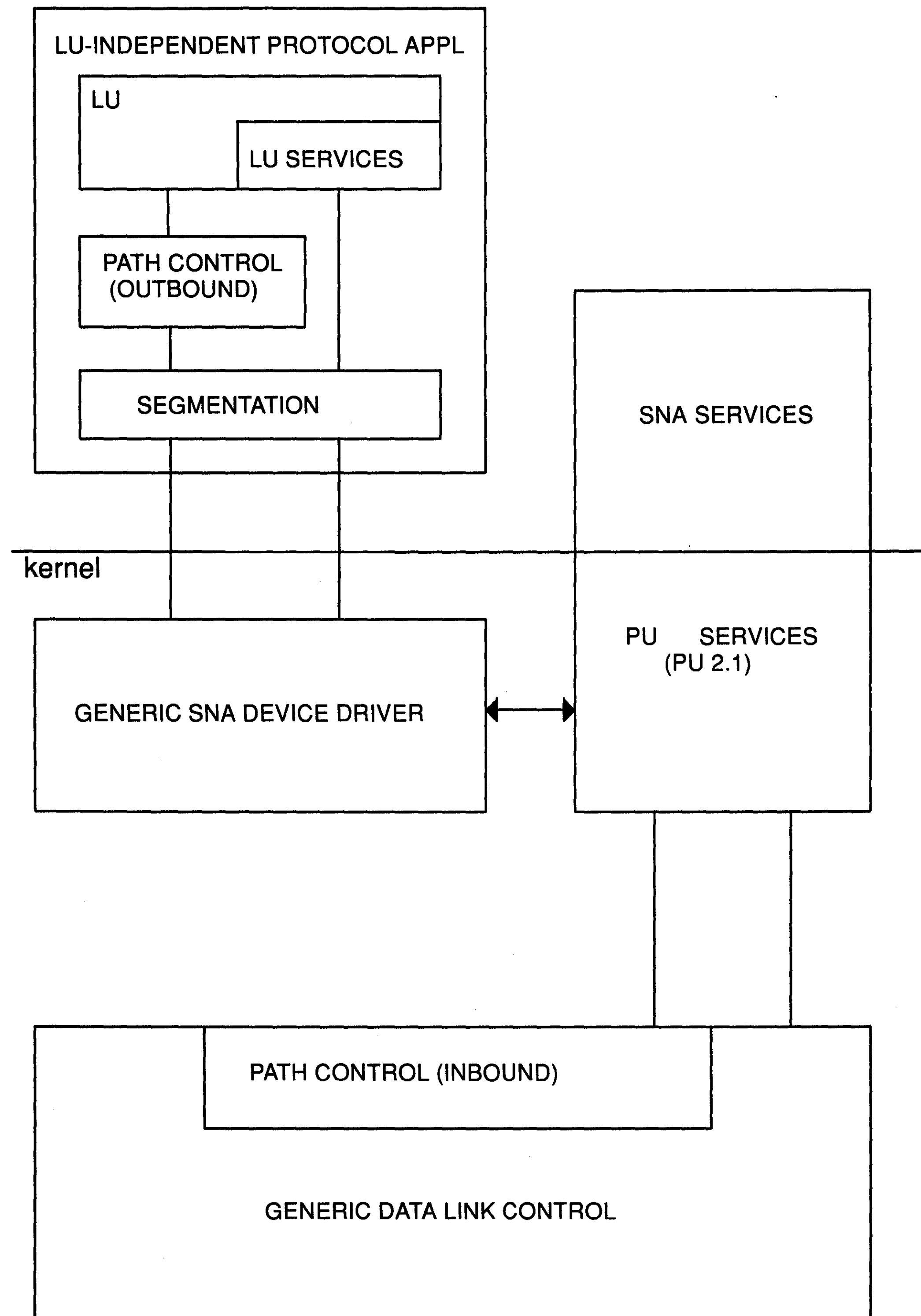
Refer to *Systems Network Architecture: Technical Overview (GC30–3073)* for further information concerning the generic SNA device driver.

Overview

AIX SNA Services/6000 currently provides support for PU type 2.1 and LU types 1, 2, 3, and 6.2. If additional LU SNA function support is required on AIX SNA Services/6000 (such as LU type 0), the user can write personalized SNA application code to interface with the generic SNA device driver.

The generic SNA device driver provides support that allows the generic SNA application code to use the PU Services of AIX SNA Services/6000. This is done through the use of the **open** (specify attachment profile name), **close**, **read**, **write**, **select**, and **ioctl** subroutines for Generic SNA. This function support allows the user to write LU-independent protocol application on top of the AIX SNA Services/6000 PU 2.1 node.

The following figure shows the structure of the LU-independent protocol application running on top of the PU Services of AIX SNA Services/6000.



Configurations

The generic SNA device driver is part of the AIX SNA Services/6000 licensed program. After installation, it will reside in the AIX kernel. The generic SNA device driver supports multiple application processes and runs under the application process control. The appropriate device driver and data link control must be installed and configured on the system.

For generic SNA device drivers, users are required to provide their own utilities to configure the SNA Logical Unit and network information.

The local LU address(es) used by the generic SNA application need to be registered in an address registration profile, which is specified in the attachment profile. This registration can be done using either the SMIT Interface described in Customizing AIX SNA Services/6000 or the configuration commands listed under AIX SNA Services/6000 Commands in *Commands Reference*

Functional Characteristics

Generic SNA Device Driver Interface To AIX SNA Services

The generic SNA device driver provides a special interface to AIX SNA Services/6000. This special interface allows the generic SNA application to utilize the PU Services function (PU 2.1 only) of AIX SNA Services/6000.

The generic SNA application can use this special interface to establish an AIX SNA Services/6000 attachment or share the same AIX SNA Services/6000 attachment with AIX SNA Services/6000. For example, a generic SNA LU_0 secondary application can run simultaneously with a 3270 emulation application to the Host system over the same AIX SNA attachment. Other programming interfaces for the following subroutines are described in detail in AIX SNA Services/6000 Subroutines in *Calls and Subroutines Reference.*

The generic SNA device driver provides the following API subroutines to interface with AIX SNA Services/6000:

open Specifies an AIX SNA Services/6000 attachment profile name in the open path to open a file descriptor for an AIX SNA Services/6000 attachment.

close Closes a file descriptor.

read Receives data from a file descriptor.

write Sends data to a file descriptor.

select Waits for file descriptors until data is available or until an exception condition occurs for the file descriptors.

ioctl Performs the following command options:

HIER_RESET_RSP	Responds to a hierarchical reset from the PU Services of AIX SNA Services/6000.
INOP_RSP	Responds to an INOP from the PU Services of AIX SNA Services/6000.
IOCINFO	Requests the type of device.

The following figure shows the structure of the relationship between the generic SNA device driver and AIX SNA Services/6000.

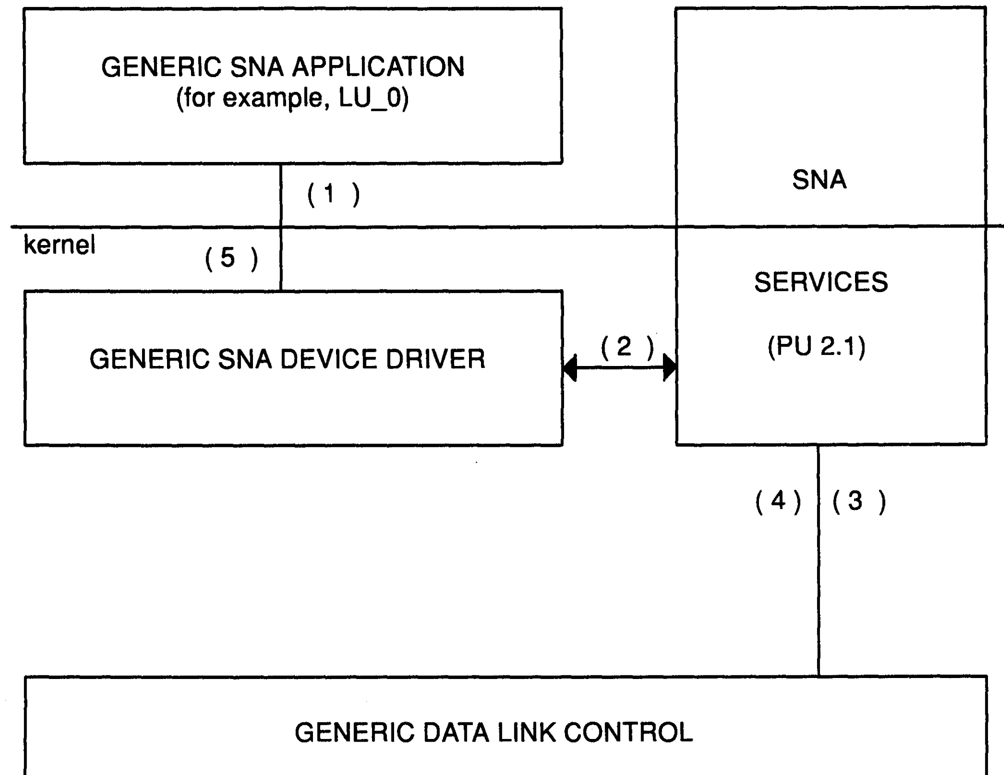


Figure 1. Relationship between Generic SNA Device Driver and AIX SNA Services/6000

1. The generic SNA application issues the **open** (attachment profile name) subroutine to the generic SNA device driver.
2. The generic SNA device driver invokes AIX SNA Services/6000 to start an attachment.
3. AIX SNA Services/6000 interfaces with the generic data link control to start the attachment. AIX SNA Services/6000 also passes LU addresses (registered for the generic SNA device driver) to Path Control.
4. When Path Control receives data, it routes the PIU to AIX SNA Services/6000 if the LU address is not registered. If the LU address is registered, Path Control routes the PIU to the generic SNA device driver.
5. The generic SNA device driver returns the PIU to the generic SNA application.

Operator Interface

There is no special operator interface for the generic SNA device driver. Operator commands (**startsrc**, **stopsrc**, **lssrc**, and so forth) for AIX SNA Services/6000 may be used.

Application Program Interface

The programming interface defined here is used by a process to issue subroutines directly to the generic SNA device driver. The following AIX subroutines are used: **open**, **close**, **read**, **write**, **select**, and **ioctl**.

AIX SNA Services/6000 Terminology

ACTPU	Activate Physical Unit
AIX	Advanced Interactive Executive Operating System
API	Application Program Interface
CPS	Control Point Services
CR	Command Router
DACTPU	Deactivate Physical Unit
GDLC	Generic Data Link Control
IODN	Input/Output Device Number
INOP	Inoperative
IPC	Inter-Process Communication
LNS	LU Network Services
PC	Path Control
PSB	Program Status Block
PU	Physical Unit
RH	Request/Response Header
RISC	Reduced Instruction Set Computing
RU	Request/Response Unit
SAP	Service Access Point
SNA	Systems Network Architecture
SVC	Supervisor Call
GSNA_DD	Generic SNA Device Driver
SNA_MDD	SNA Manager Device Driver
TH	Transmission Header
XID	Exchange Identification.

IBM AIX SNA Services/6000 LU0 Facility

Document Guide

Intended Readership

This document is intended for those persons interested in using the LU0 Subsystem on the RISC System/6000.

Reader Preconditions

The reader should have a basic understanding of the RISC System/6000, the AIX operating system, SNA concepts, and AIX SNA Services/6000.

Reader Postconditions

After completing this document, the reader should:

- Have an understanding of the use of the LU0 support and its relationships with AIX SNA Services/6000.

Recommended Reference Documentation

Refer to the following documents for further information concerning LU0 support.

- *AIX Systems Network Architecture Services/6000 Guide and Reference*
- *Calls and Subroutines Reference*
- *Files Reference*
- *IBM Advanced Data Communication for Stores Program Reference and Operation Guide*
- *SNA Technical Overview.*

Introduction

This document provides the information necessary to configure, start, and stop the LU0 subsystem. The application program interface is also defined in this document. The following subjects are covered in detail in this manual:

Configuring the LU0 Subsystem
This section guides the user through the steps necessary to configure the LU0 subsystem. Use of SNA Services for Secondary support and the LU0 Configurator for Secondary and Primary support are covered.

Using the LU0 Subsystem
This section covers starting, stopping, and command line arguments for the LU0 Subsystem.

API This section gives a description of how the LU0 Subsystem provides an application program interface for the Primary and Secondary support. These subroutines allow you to open, send, receive, close, and control the session from within your application. The Secondary and Primary support have separate APIs for use by the application program.

Applications This section provides three applications as part of the LU0 Subsystem: the Advanced Data Communications for Stores (ADCS) Emulator, the Host Command Processor (HCP) Emulator, and the LU0 Passthrough facility. This section discusses these applications and their use.

Using the Menus

When the main menu appears, you can select any of the options displayed by entering the number of the option:

```
—————————————————— MAIN MENU ——————————————————

                1. Define LU0 Secondary.
                2. LU0 Primary.
                3. Print Configuration File.
                4. Exit.

        Enter Option Number :

F3=exit program
```

The available options are:

Field Name **Description**

`Define LU0 Secondary`
See Defining LU0 Secondary Support on page 8–38.

`Define LU0 Primary`
See Defining LU0 Primary Support on page 8–40.

`Print Configuration file`
This option prints a formatted report showing the Primary LU0 and Secondary LU0 definitions. This report is written to the **/usr/lpp/lu0/lu0confg.rpt** file.

`Exit` This option exits the configuration utility.

Function key:

`F3` Exits. This option exits the configuration utility.

Defining LU0 Secondary Support

Defining the LU0 Secondary Support entails defining the logical units to be used for Secondary support. AIX SNA Services/6000 is required for Secondary support. The following menu shows the parameters you specify to define the LU0 Secondary LUs:

```
———————— LU0 SECONDARY SPECIFICATIONS ————————
        LU Name       =====>    (REQUIRED ENTRY)
        Send Initself =====>    (Y or N)
        LU Address    =====>    (1–255)
        Host Appl Name ====>    (for init_self only)
        Log Mode Table Ent=>    (for init_self only)
        Passthru Partner ==>    (Primary LU name for passthru)
        Enable API Trace ==>    (Y or N)

F2=clear  F3=exit  F5=refresh  F6=write file
F7=next record  F9=delete record
```

Field Name **Description**

`LU Name` KEY - Required

The name, 1 to 8 alphanumeric characters, of the logical unit. The LU Name corresponds to an SNA attachment name.

`Send Initself`
: Optional

  Enter `Y` if this LU should send an INIT-SELF request to the host. Enter `N` if the secondary LU should wait for a BIND request from the host.

`LU Address`
: Required

  This numeric value must correspond to a previously defined local LU address on the attachment profile dialog. No other secondary LU entries in the file may use this LU address.

`Host Appl Name`
: Optional

  The name of the host application to use as a session partner.

`Log Mode Table Ent`
: Optional

  If this field is specified, the log mode entry name will be specified in the INIT-SELF request.

`Passthru Partner`
: Optional

  If this field is specified, this primary LU on the RISC System/6000 is the passthru partner and all PIUs received are sent to this PLU. If you specify `Init-Self = Y`, then you may not enter a value in this field.

`Enable API Trace`
: Optional

  Enter a `Y` to enable this option. If this field is specified, a trace file is written when an application reads from, or writes data to, this device. The trace file specification is: **/usr/lpp/lu0/***LUName*, where the *LUName* parameter specifies the logical unit name for this device (same as LU Name entry above). Please refer to the **lu0** Command for more information on this file.

Function keys	**Description**
`F2`	Erases the unprotected fields of this screen. All unsaved changes are lost.
`F3`	Exits. Returns to the main menu. Unsaved changes are lost.
`F5`	Refreshes the screen from the currently saved file. Use the LU name to determine which record to display. Unsaved changes are lost.
`F6`	Writes this record to file. Only this record is used to update the file.
`F7`	Displays the next sequential secondary record in the file. When end-of-file is reached, the first secondary record in the file becomes current and is displayed. Unsaved changes are lost.
`F9`	Deletes the current secondary record.

Defining LU0 Primary Support

Defining the LU0 Primary Support entails of defining the port, physical unit, application logical units, session partners, xid tables, and log mode entries. The following menu shows the parameters you specify to define the LU0 Primary Support:

```
---------------  LU0  PRIMARY  LINE  ------------------

        Port Name        =====>  (mpq 0-7)
        Physical Unit Name ==>  (optional)
        Max data         =====>  (Max i-field size, 89-4100)
        Remote Station Addr==>  (01-FE)
        Local Station Addr ==>  (01-FE)
        Modem            =====>  (1=NRZ 2=NRZI)
        Connect          =====>  (1=Switched     2=Non-switched)
        RTS              =====>  (1=Controlled   2=Continuous)
        DTR              =====>  (1=DTR          2=CDSTL)

F3=exit   F5=refresh   F6=write record
F8=define Primary records
```

Field Name **Description**

`Port Name` Required

The name, 1 to 8 alphanumeric characters, of the SDLC port. This is used as an identifier only.

`Physical Unit Name`
Required

The name of the physical unit to be defined on this port.

`Max Data` Default = 265

The size of the maximum I-field, 89 to 4100 bytes, that the secondary PU can receive or send.

`Remote Station Addr`
Required

The SDLC station address, hex 01 to FE, that this PU will be calling.

`Local Station Addr`
Required

The SDLC station address, hex 01 to FE, for polling this PU.

`Modem` Required

Enter `1` for NRZ. Enter `2` for NRZI.

`Connect` Required

Enter `1` for a switched connection.
Enter `2` for a non-switched connection.

`RTS` Required

Enter `1` for Controlled.
Enter `2` for Continuous.

`DTR` Required

Enter `1` for DTR.

Enter 2 for CDSTL.

Function keys	Description
F3	Exits. Returns to the main menu. Unsaved changes are lost.
F5	Refreshes the screen from the currently saved file. Unsaved changes are lost.
F6	Writes this record to file. Only this record is used to update the file.
F8	Transfers control to the LU0 Primary Application Logical Units screen. Unsaved changes are lost.

Defining LU0 Primary Application Logical Units

```
—— LU0 Primary Application Logical Units ——

        LU Name        =====>    (REQUIRED ENTRY)
        PLU Address      ===>    (always 1)
        SLU Address     ====>    (1–255)
        FM Profile      ====>    (3–4)
        TS Profile      ====>    (3–4)
        FM Pri Protocols ==>     (bind byte 4)
        FM Sec Protocols ==>     (bind byte 5)
        FM Com Protocols ==>     (bind bytes 6–7)
        TS Protocols    ====>    (bind bytes 8–13)
        PS Profile      ====>    (bind bytes 14)
        PS Protocols    ====>    (bind bytes 15–25)
        User Data       ====>
        Line Name       ====>    (Optional)
        Enable API Trace ==>     (Y or N)

F2=clear   F3=exit   F5=refresh   F6=write file
F7=next record   F9=delete record
```

Fieldname	Description
LU Name	KEY-Required The name of the logical unit, 1 to 8 alphanumeric characters in length.
PLU Address	Protected field: always set to 1 The LU number of the primary LU.
SLU Address	Required The LU number, 1 to 255, of the secondary LU.
FM Profile	Required The Function Management profile that the session uses. FM profiles 3 and 4 are supported.
TS Profile	Required The Transmission Subsystem profile that the session uses. TS profiles 3 and 4 are supported.

`FM Pri Protocols`
Required

The primary logical unit protocols, 2 hex digits in length, that the session uses for FM data.

`FM Sec Protocols`
Required

The secondary logical unit protocols, 2 hex digits in length, that the session uses for FM data.

`FM Com Protocols`
Required

The logical unit protocols, 4 hex digits in length, and common to both the primary and secondary logical units that the session uses for FM data.

`TS Protocols`
Required

The Transmission Subsystem protocols, 12 hex digits in length, that the session uses.

`PS Profile` Required

The PS profile, 2 hex digits in length, that the session uses.

`PS Protocols`
Default: 0 hex

The PS protocols that the session uses. Specify an even number of digits, up to 22 digits in length.

`User Data` Optional

Any hexadecimal user data that you want to send in the BIND command. Specify an even number of hex digits, up to 48 digits in length.

`Line Name` Required

The name of the SDLC line, 1 to 8 alphanumeric characters in length. It should match the Port Name field on the LU0 Primary Line Definitions screen.

`Enable API Trace`
Optional

Enter a `Y` to enable this option. If this field is specified, a trace file is written when an application reads from, or writes data to, this device. The trace file specification is the **/usr/lpp/lu0/***LUName*, where the *LUName* parameter specifies the logical unit name for this device (same as LU Name entry above). Please refer to the **lu0** Command for more information on this file.

Function Keys Definition

`F2`	Erases the unprotected fields of this screen. All unsaved changes are lost.
`F3`	Exits. Returns to the Define LU0 Primary screen. Unsaved changes are lost.
`F5`	Refreshes the screen from the currently saved file. Use the LU name to determine which record to display. Unsaved changes are lost.
`F6`	Writes this record to file. Only this record is used to update the file.

F7 Displays the next sequential primary record in the file. When end-of-file is reached, the first primary record in the file becomes current and is displayed. Unsaved changes are lost.

F9 Deletes the current primary record. Unsaved changes are lost.

Application Program Interface

The LU0 Subsystem provides an application program interface for Primary and Secondary support. These subroutines allow you to open, send, receive, close, and control the session from within your application.

The Secondary and Primary support have separate APIs for use by the application program.

Secondary LU0 API

Five subroutines are available in the Secondary API. They are the **lu0closes, lu0ctls, lu0opens**, **lu0reads**, and **lu0writes** subroutines.

Primary LU0 API

Five subroutines are available in the Primary API. They are the **lu0closep, lu0ctlp, lu0openp**, **lu0readp**, and **lu0writep** subroutines.

Applications

Three applications are provided as part of the LU0 Subsystem: the Advanced Data Communications for Stores (ADCS) Emulator, the Host Command Processor (HCP) Emulator, and the lu0 Passthrough API.

Advanced Data Communications for Stores (ADCS) API

The Advanced Data Communications for Stores (ADCS) API uses the **lu0api** subroutine to create the **cmd4680** file used by the ADCS emulator. You must compile the **lu0api.c** module with your source code.

Host Command Processor (HCP) API

The Host Command Processor (HCP) API uses the **hcp** and **stophcp** commands to start and stop the Host Command Processor. The HCP API also uses the **hcps** and **stophcps** commands to start and stop the HCP SUP Command Processor.

lu0 Passthrough API

The lu0 Passthrough API uses the **lu0pass** command to start the lu0 Passthrough application, which provides LU0 connectivity on an AIX RISC System/6000 machine between a host and a 4680 Store Controller.

Writing Transaction Programs for AIX SNA Services/6000

Writing Transaction Programs for AIX SNA Services/6000 provides information to help you write an application program that uses AIX SNA Services/6000 to transfer data over a network. Such a program is called a *transaction program*. The information is divided into two parts. The first part provides guidelines for interfacing the program to AIX SNA Services/6000, including the parameters passed to all transaction programs and the signals sent to programs. The second part provides simplified examples of a local and a remote transaction program to illustrate the use of the AIX SNA Services/6000 application programming interface.

Guidelines for Writing Transaction Programs

An application program that uses AIX SNA Services/6000 to transfer information over a network must conform to the interface conventions that AIX SNA Services/6000 uses. The following paragraphs outline those conventions.

Parameter Passing

When AIX SNA Services/6000 starts a remote transaction program, it passes information to that program using the *argv*[ ] parameters of a C language program. How this information is passed depends upon whether the application program uses the limited interface or the extended interface.

Using the Extended Interface

When a program uses the extended interface, AIX SNA Services/6000 passes the following information to the program when it starts:

Parameter	Contents
argv[0]	The name of the transaction program.
argv[1]	The name of the TPN profile for the transaction program.
argv[2]	The name of the connection profile for the transaction program.
argv[3]	The resource ID of the connection in string format. Use the library function **a64l** with this parameter to get the resource ID: `resource_id = a64l( argv[3] );`
argv[4], *argv*[5], ..., *argv*[*n*]	If the remote transaction program receives PIP (program initialization parameter) data, these arguments contain the subfields of that PIP data.

Using the Limited Interface

When a program uses the limited interface, AIX SNA Services/6000 passes the following information to the program when it starts:

Parameter	Contents
argv[0]	The name of the transaction program.

argv[1] The name of the TPN profile for the transaction program.

argv[2] The name of the connection profile for the transaction program.

The limited interface does not support PIP data and does not require a resource ID.

Signals

AIX SNA Services/6000 sends the signal `SIGUSR1` to all transaction programs, both local and remote, when one of the following conditions occurs:

- The connection is stopped.
- AIX SNA Services/6000 is stopped.

If a transaction program receives this signal, it must close the file descriptor for the connection.

When the `SIGUSR1` signal is sent to a process, it is also directed to other processes that have the same process group ID.

For example, assume that the command interpreter **bsh** started an application process that interfaces with AIX SNA Services/6000. A `SIGUSR1` signal sent to the application process is also sent to the command interpreter and may stop the interpreter process. To avoid this, an application process that will run as a background task can change its process group ID by using the **setpgrp** subroutine when it begins execution.

AIX SNA Services/6000 Example Programs

The following pages contain short example programs that illustrate the use of many of the subroutines described in this manual to control operation of a network using AIX SNA Services/6000. The programs have been abbreviated by leaving out most error checking and recovery procedures that would normally be included in programs of this type. Although the programs do compile and run, they are included as examples only and should not be used for actual data operations without significant changes to provide the missing functions.

These programs illustrate programs running at each end of the conversation:

- "Local Transaction Program Example Program" on page 8–46 shows the program that starts the conversation.
- "Remote Transaction Program Example Program" on page 8–48 shows the program that responds to the conversation.
- "Mapped Transaction Program Example" on page 8–51 shows a program that starts a mapped conversation.
- "Mapped Remote Transaction Program Example" on page 8–53 shows the program that responds to the mapped conversation.

Local Transaction Program Example Program

The example program shown in the following example illustrates the use of subroutines by a transaction program that invokes a partner transaction program on a remote logical unit. This program performs the following actions:

1. Opens a connection.
2. Allocates a conversation with the remote transaction program, **sysrmt**.
3. Sends data to the remote program.
4. Receives data from the remote program.
5. Closes the connection.

The input parameter, `argv[1]`, is the name of the connection profile for the remote LU. The program is started by entering the following command:

```
$ sysloc cpn
```

Where the *cpn* parameter is the name of the connection profile for the remote LU.

When the program runs, it prints messages to standard output that describe the events that occur, such as subroutines and data transmissions. To make the program easier to understand, its only reaction to an error is to exit and set **return_code** to the error value. The following major events occur during the programs operation:

1. Opens the resource whose path name consists of the device driver name, followed by the connection profile name, which was given as an argument to the program.
2. Initializes the **allo_str** structure to nulls and then fills in the remote transaction program name. Leaves other fields null for default values.
3. Issues the **ioctl** subroutine to allocate the conversation and get a resource ID. When the allocation is complete, the program is in send state.
4. Initializes the **ext_io_str** structure to nulls and fills in the resource ID to prepare for the **readx** or **writex** subroutine.
5. Sends data to the remote transaction program, using the **writex** subroutine. This data is prefixed by byte count and **gdsid**. Once the data is sent, the program remains in the send state.
6. Issues the **readx** subroutine to flush the send buffer, put the program into the receive state, and read data from the remote transaction program.
7. Prints out the information received. A value is printed indicating what control information was received; this value should be 5 for normal deallocate, indicating that the remote program issued a DEALLOCATE request with type parameter of FLUSH.

8. Issues the **close** subroutine to close the connection.

```
#include     <stdio.h>
#include     <fcntl.h>
#include     <errno.h>
#include     <luxsna.h>

#define NULL    0
#define ERROR  -1
#define OK      0

extern  int     errno;
main( int     argc,               /*number arguments received      */
      char  **argv )              /*pointers to arguments received */
{
    char    buf[80];              /*buffer for input data    */
    char    path[80];             /*buffer for pathname of resource*/
    int     fd;                   /*file descriptor for connection */
    int     nbytes;               /*number of bytes read or written*/
    int     resource_id;          /*resource ID of conversation    */
    int     return_code = OK;     /*exit code                      */
    struct  allo_str  allo_str;   /*structure for ioctl(ALLOCATE)  */
    struct  ext_io_str ext_str;   /*structure for readx subroutine */
    strcpy( path, "/dev/sna" );
    strcat( path, argv[1] );
    if ( ( fd = open( path, O_RDWR ) ) == ERROR ) {
        return_code = errno;
    }
    else {
        printf( "Connection established with < %s >.\n", path );
        memset( &allo_str, 0, sizeof( struct allo_str ) );
        strcpy( allo_str.tpn, "sysrmt" );
        if ( ioctl( fd, ALLOCATE, &allo_str ) == ERROR ) {
            return_code = errno;
        }
        else {
            resource_id = allo_str.rid;
            printf("Conversation allocated with tpn < %s >.\n",
                     allo_str.tpn);
            buf[0] = 0x00;
            buf[1] = 0x1c;
            buf[2] = 0x12;
            buf[3] = 0xff;
            memset( &ext_str, 0, sizeof( struct ext_io_str ) );
            ext_str.rid = resource_id;
            strcpy( &buf[4], "Please send me a message" );
            if ( ( nbytes = writex( fd, buf, 29, &ext_str ) ) ==
                        ERROR ) {
                return_code = errno;
            }
            else {
                printf( "Bytes sent: < %d >\n", nbytes );
                printf( "Data sent: < %s > \n", &buf[4] );
                if ( ( nbytes = readx( fd, buf, 29, &ext_str ) ) ==
                        ERROR ) {
```

```
                    return_code = errno;
                }
                else {
                  printf( "Bytes received: < %d >\n", nbytes );
                  printf( "Data received: < %s > \n", &buf[4] );
                  printf( "Control information received: < %s > \n",
                        ext_str.output.what_control_rcvd );
                    close( fd );
                    printf( "Conversation deallocated.\n" );
                    printf( "Connection closed.\n" );
                }
            }
        }
    }
    exit( return_code );
}
```

Remote Transaction Program Example Program

The example program shown in the following example illustrates the use of subroutines by a transaction program that was invoked by a partner transaction program on a remote logical unit. This program performs the following actions:

1. Opens a connection, using the connection profile it receives as an argument.
2. Allocates a conversation with the remote transaction program, **sysrmt**, resource ID it receives as an argument to the program.
3. Issues the **ioctl** subroutine with a `REQUEST_TO_SEND` request to the remote program.
4. Receives control information and data from the remote program, using the **readx** subroutine.
5. Sends data to the remote program, using the **writex** subroutine.
6. Deallocates the conversation.
7. Closes the connection.

The input parameter, *argv*[2], is the name of the connection profile for the remote LU. The input parameter, *argv*[3], is the resource ID for the connection.

When the program runs, it prints messages to standard output that describe the events that occur, such as subroutines and data transmissions. To make the program easier to understand, its only reaction to an error is to exit and set **return_code** to the error value. The following major events occur during the programs operation:

1. Opens the resource whose path name consists of the device driver name (**/dev/sna**) followed by the connection profile name. The connection profile name is an argument to the program.
2. Initializes the **allo_str** structure to nulls to select the default values.
3. Converts the rid that was received as a string argument (*argv*[3]) into an integer. Puts the integer into the `rid` field of `allo_str` to indicate a remote attach.
4. Issues the **ioctl** subroutine with an `ALLOCATE` request, leaving this program in the receive state.

5. Issues the **ioctl** subroutine with a `REQUEST_TO_SEND` request to notify the remote program that this program needs to send data. This program remains in the receive state.
6. Reads data and control information from the remote program, using the **readx** subroutine.
7. Prints out the information received. A value is printed indicating what control information was received; this value should be 1 for the *send* parameter, indicating that the remote program is in the receive state and that this program is now in the send state.
8. Sends data to the `remote` program, using the **writex** subroutine. This data is prefixed by byte count and **gdsid**. Once the data is sent, the program remains in the send state.
9. Fills in the `deal_str` structure fields with:

   `A type of DEAL_FLUSH`
   : To flush the send buffer and deallocate the conversation normally

   `A flag of DISCARD`
   : To discard the conversation when it is deallocated.
10. Issues the **ioctl** subroutine with a `DEALLOCATE` request to deallocate the conversation.
11. Issues the **close** subroutine to close the connection.

```
#include    <stdio.h>
#include    <fcntl.h>
#include    <errno.h>
#include    <luxsna.h>

#define NULL           0
#define ERROR         -1
#define OK  0

extern  int     errno;
extern  int     a641();
main( int     argc,            /* number arguments received      */
      char   **argv )          /* pointers to arguments received */
{
    char    buf[80];           /* buffer for input data          */
    char    path[80];          /* buffer for pathname of resource*/
    int     fd;                /* file descriptor for connection */
    int     nbytes;            /* number of bytes read or written*/
    int     resource_id;       /* resource ID of conversation    */
    int     return_code = OK; /* exit code                       */
    struct  allo_str  allo_str;/* structure for ioctl(ALLOCATE)  */
    struct  deal_str deal_str; /* structure for ioctl(DEALLOCATE)*/
    struct  ext_io_str ext_str;/* structure for readx subroutine */
    strcpy( path, "/dev/sna" );
    strcat( path, argv[2] );
    if ( ( fd = open( path, O_RDWR ) ) == ERROR ) {
        return_code = errno;
    }
    else {
```

```
    printf( "Connection established with < %s >.\n", path );
    memset( &allo_str, 0, sizeof( struct allo_str ) );
    allo_str.rid = a641( argv[3] );
    if ( ioctl( fd, ALLOCATE, &allo_str ) == ERROR ) {
        return_code = errno;
    }
    else {
        resource_id = allo_str.rid;
        printf("Conversation allocated (remote attach):  rid
                = < %d >.\n",resource_id );
        if ( ioctl( fd, REQUEST_TO_SEND, resource_id ) == ERROR
                ) {
            return_code = errno;
        }
        else {
            printf("REQUEST TO SEND ioctl issued.\n");
            memset( &ext_str, 0, sizeof( struct ext_io_str ) );
            ext_str.rid = resource_id;
            if ( ( nbytes = readx( fd, buf, 29, &ext_str ) ) ==
                    ERROR ) {
                return_code = errno;
            }
            else {
              printf( "Bytes received: < %d >\n", nbytes );
              printf( "Data received: < %s > \n", &buf[4] );
              printf( "Control information received: < %s > \n",
                ext_str.output.what_control_rcvd );
               buf[0] = 0x00;
               buf[1] = 0x1d;
               buf[2] = 0x12;
               buf[3] = 0xff;
              strcpy( &buf[4], "This is a message for you." );
              if ( (nbytes = writex( fd, buf, 29, &ext_str )) ==
                        ERROR ) {
                    return_code = errno;
                }
                else {
                    printf( "Bytes sent: < %d >\n", nbytes );
                    printf( "Data sent: < %s > \n", &buf[4] );
                    deal_str.rid = resource_id;
                    deal_str.type = DEAL_FLUSH;
                    deal_str.deal_flag = DISCARD;
                    if ( ioctl( fd, DEALLOCATE, &deal_str ) ==
                            ERROR ) {
                        return_code = errno;
                    }
                    else {
                        close( fd );
                        printf( "Conversation deallocated.\n" );
                        printf( "Connection closed.\n" );
                    }
                }
            }
        }
    }
}
```

```
    exit( return_code );
}
```

Mapped Local Transaction Program Example Program

The example program shown in the following example illustrates the use of subroutine calls running on a mapped conversation by a transaction program that invokes a partner transaction program on a remote logical unit.

```
#include    <stdio.h>
#include    <string.h>
#include    <fcntl.h>
#include    <errno.h>
#include    <luxsna.h>

#define NULL    0
#define ERROR   -1
#define OK      0

extern  int     errno;
main( int     argc,            /* number arguments received        */
      char  **argv )           /* pointers to arguments received   */
{
    char    buf[80];           /* buffer for input data            */
    char    path[80];          /* buffer for pathname of resource  */
    char    c_type = 'M';      /* Conversation type -- mapped      */
    int     connection_id;     /* file descriptor for connection   */
    int     length;
    int     nbytes;            /* number of bytes read or written  */
    int     resource_id;       /* resource ID of conversation      */
    int     return_code = OK;  /* exit code                        */
    struct  allo_str    allo_str;/* structure for ioctl(ALLOCATE) */
    struct  read_out    read_out;/* structure for readx subroutine*/
    struct  write_out   write_out;
    strcpy( path, argv[1] );
    printf("Path = < %s >.\n", path);
    if ( ( connection_id = snaopen( path ) ) == ERROR ) {
        printf("snaopen failed, errno = < %d >.\n", errno);
        return_code = errno;
    }
    else {
        printf( "Connection established with < %s >.\n", path );
        memset( &allo_str, 0, sizeof( struct allo_str ) );
        strcpy( allo_str.tpn, "sysrmtmap" );
        if ( ( resource_id = snalloc( connection_id, &allo_str,
                c_type ) ) == ERROR ) {
            printf("snalloc failed, errno = < %d >.\n", errno);
            return_code = errno;
        }
        else {
            printf("Conversation allocated with tpn < %s >.\n",
                    allo_str.tpn);
            length = 24;
            strncpy( buf, "please send me a message", length );
```

```
                if ( ( nbytes = snawrit( connection_id, buf, length,
                        resource_id, &write_out, c_type ) ) == ERROR ) {
                    printf("snawrit failed, errno = < %d >.\n", errno);
                    return_code = errno;
                }
                else {
                    printf( "Bytes sent: < %d >\n", nbytes );
                    printf( "Data sent: < %s > \n", buf );
                    length = 80;
                    if ( ( nbytes = snaread( connection_id, buf, length,
                     resource_id, 0, &read_out, c_type ) ) == ERROR
OR                            ) {
                        printf("snaread failed, errno = < %d >.\n",
                                errno);
                        return_code = errno;
                    }
                    else {
                        printf( "Bytes received: < %d >\n", nbytes );
                        printf( "Data received: < %s > \n", buf );
                      printf( "Control information received: < %s > \n",
                            read_out.what_control_rcvd );
                    }
                }
            }
        }
        exit( return_code );
}
```

Mapped Remote Transaction Program Example Program

The example program shown in the following example illustrates the use of subroutine calls running on a mapped conversation by a transaction program that was invoked by a partner transaction program on a remote logical unit.

```
#include    <stdio.h>
#include    <fcntl.h>
#include    <errno.h>
#include    <luxsna.h>

#define NULL    0
#define ERROR   -1
#define OK      0

extern  int     errno;
extern  int     a641();
main( int     argc,            /* number arguments received       */
      char  **argv )           /* pointers to arguments received  */
{
    char    buf[80];           /* buffer for input data           */
    char    path[80];          /* buffer for pathname of resource */
    char    c_type = 'M';      /* Conversation type — mapped      */
    int     connection_id;     /* file descriptor for connection  */
    int     length;
    int     nbytes;            /* number of bytes read or written */
    int     resource_id;       /* resource ID of conversation     */
    int     return_code = OK; /* exit code                        */
    struct  allo_str     allo_str;/* structure for ioctl(ALLOCATE) */
    struct  deal_str     deal_str;/*structure for ioctl(DEALLOCATE)*/
    struct  read_out     read_out;/* structure for readx subroutine*/
    struct  write_out    write_out;
    strcpy( path, argv[2] );
    if ( ( connection_id = snaopen( path ) ) == ERROR ) {
        printf("snaopen failed, errno = < %d >.\n", errno);
        return_code = errno;
    }
    else {
        printf( "Connection established with < %s >.\n", path );
        memset( &allo_str, 0, sizeof( struct allo_str ) );
        allo_str.rid = a641( argv[3] );
        if ( ( resource_id = snalloc( connection_id, &allo_str,
                c_type ) ) == ERROR ) {

            printf("snalloc failed, errno = < %d >.\n", errno);
            return_code = errno;
        }
        else {
            printf("Conversation allocated (remote attach):  rid =
                    < %d >.\n", resource_id );
            if ( snactl( connection_id, REQUEST_TO_SEND,
                    resource_id, c_type ) == ERROR ) {
```

```
                printf("REQUEST TO SEND failed, errno = < %d >.\n",
                        errno);
                return_code = errno;
            }
            else {
                printf("REQUEST TO SEND snactl issued.\n");
                length = 80;
                if ( ( nbytes = snaread( connection_id, buf, length,
                        resource_id, 0, &read_out, c_type ) ) ==
                        ERROR ) {
                    printf("snaread failed, errno = < %d >.\n",
                            errno);
                    return_code = errno;
                }
                else {
                    printf( "Bytes received: < %d >\n", nbytes );
                    printf( "Data received: < %s > \n", buf );
                  printf( "Control information received: < %s > \n",
                        read_out.what_control_rcvd );
                    length = 26;
                    strncpy( buf, "This is a message for you.",
                            length );
                    if ( ( nbytes = snawrit( connection_id, buf,
                         length, resource_id, &write_out, c_type ) )
                            == ERROR ) {
                        printf("snawrit failed, errno = < %d >.\n",
                                errno);
                        return_code = errno;
                    }
                    else {
                        printf( "Bytes sent: < %d >\n", nbytes );
                        printf( "Data sent: < %s > \n", buf );
                        deal_str.rid = resource_id;
                        deal_str.type = DEAL_FLUSH;
                        deal_str.deal_flag = DISCARD;
                        if ( snadeal( connection_id, &deal_str,
                                c_type ) == ERROR ) {
                            return_code = errno;
                        }
                        else {
                            snaclse( connection_id );
                            printf( "Conversation deallocated.\n" );
                            printf( "Connection closed.\n" );
                        }
                    }
                }
            }
        }
    }
    exit( return_code );
}
```

Transferring Files Using AIX SNA Services/6000

Transferring Files Using AIX SNA Services/6000 describes the file transfer programs included with AIX SNA Services/6000.

Two sample programs, **sendto.c** and **rcvfrom.c**, send files from a local AIX node to a remote AIX node, using SNA LU 6.2 protocol. To use these programs, both nodes must have the proper communication adapters installed. In addition, both nodes must have the software for AIX and SNA, which includes the file transfer programs.

During a mapped conversation, the local AIX node can start a file transfer by using the **sendto** command. This command activates the **rcvfrom** program on the remote or partner AIX node so the partner can receive the files.

Another time the roles can be reversed, with the partner AIX node becoming the system that initiates the file transfer.

The file transfer programs are in the **samples** directory, and have the following path: `/usr/lpp/sna/samples.`

To compile the programs, use the following commands:

```
cc sendto.c -o sendto
cc rcvfrom.c -o rcvfrom -lsna
```

To start a file transfer, give the following command on the local system, substituting the actual names for the connection and the file names:

`sendto` *`ConnectionName LocalFileName RemoteFileName`*

If the *RemoteFileName* parameter is not specified, the **sendto.c** program defaults to *LocalFileName* so that the user would see the following:

`sendto` *`ConnectionName LocalFileName LocalFileName`*

The following subroutines are called:

- **snaopen**
- **snalloc**
- **snaread**
- **snawrit**
- **snaclse**
- **snadeal**

For a description of these subroutines, refer to "AIX SNA Services/6000 Subroutines" on page 8-1. Parameters defined in command structures are described in "Special Files" on page 8-5.

Sending Files from a Local AIX Node to a Remote AIX Node

The **sendto.c** program includes the following steps:

sendto.c Program

1. Open the data file. (**open**)
2. Obtain the connection name and local file name. (**strcpy**)
3. Open a connection. (**snaopen**)
4. Allocate a conversation. (**snalloc**)

- Set the type of conversation that you want with the c_type parameter.

  This sample program establishes a mapped conversation between a local and a remote transaction program by setting c_type to `M`.

- Set the sync_level parameter in `*allo_str` to indicate whether you want synchronized processing between the two transaction programs.

  This program synchronizes the processing by setting `sync_level` to `confirm`.

5. Read data file into the sending buffer. (**read**)
6. Write data from the sending buffer to the receive file at the remote AIX node. (**snawrit**)

   Repeat steps 4 and 5 until EOF is read.

7. Ask the remote program to confirm when the data is received. (**snactl** with the CONFIRM request)
8. Deallocate the conversation. (**snadeal**)
9. Close the connection. (**snaclse**)

Receiving Files from a Remote AIX Node

The **rcvfrom.c** program includes the following steps:

rcvfrom.c Program

1. Obtain the connection name and resource ID. (**strcpy**)
2. Open a connection. (**snaopen**)
3. Allocate the conversation. (**snalloc**)
4. Open the data file. (**open**)
5. Read the data into the receive buffer. (**snaread**)
6. Write the data from the receive buffer to the receive file. (**write**)

   Repeat steps 5 and 6 until EOF is read.

7. Send confirmation to the sending program when the data is received. (**snactl** with the CONFIRMED request)
8. Close the connection. (**snaclse**)

Writing Generic AIX SNA Services/6000 Programs

Writing Generic AIX SNA Services/6000 Programs provides information to help you write an application that uses the generic support of AIX SNA Services/6000 to transfer data over a network. The following pages contain a short example program that illustrates the use of some of the subroutines described in this manual to control operation of a network using Generic AIX SNA Services/6000.

The program has been abbreviated by leaving out most error checking and recovery procedures that would normally be included in programs of this type. Although the program does compile and run, it is included as an example only and should not be used for actual data operations without significant changes to provide the missing functions. Note that there must also be a cooperating program running on the remote node.

Generic AIX SNA Services/6000 Example Program

The following figure illustrates the use of subroutines by a program that uses the Generic AIX SNA Services/6000 support. This program performs the following actions:

1. Opens an attachment.
2. Receives data from a remote program running on the remote node.
3. Sends data to the remote program.
4. Closes the attachment.

The input parameter, `argv[1]`, is the name of the attachment profile for the remote node. The program is started by entering the following command:

```
# gsnaloc cpn
```

Where *cpn* is the name of the attachment profile for the remote node.

When the program runs, it prints messages to standard output that describe the events that occur, such as subroutines and data transmissions. To make the program easier to understand, its only reaction to an error is to exit and set `return_code` to the error value. The following major events occur during the programs operation:

1. Opens the resource whose path name consists of the device driver name, followed by the attachment profile name, which was given as an argument to the program.
2. Issues the **read** subroutine to receive data from the remote program.
3. Prints out the information received.
4. Fills in the send buffer with the necessary control information to send data to the remote program.
5. Sends data to the remote program, using the **write** subroutine.

6. Issues the **close** subroutine to close the attachment.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/trace.h>
#include <luxgsna.h>

main(argc, argv)
int         argc;
char       *argv[];
{
    extern int errno;
    char        att_name[30];
    int         count = 0;
    int         fd;
    int         n;
    int         return_code = 0;
    char        *temp_ptr;
    char        *test_case = "secappl";    /* testcase name */
    char        buf [256];
    /*---------------------------------------------------------------*/
    /*                open to generic SNA device driver              */
    /*---------------------------------------------------------------*/
    (void) strcpy ( att_name, "/dev/gsna" );
    (void) strcat ( att_name, argv[1] );
    if ( ( fd = open( att_name, O_RDWR ) ) == -1 ) {
        printf("open to GSNA_DD failed, errno = %d\n", errno);
        return_code = errno;
    }
    else {
        /*-------------------------------------------------------------*/
        /*          Attachment is UP and RUNNING                      */
        /*-------------------------------------------------------------*/
        printf("open to GSNA_DD success, fd = %d\n", fd);
        printf("ATTACHMENT IS UP AND RUNNING....\n");
        printf("sleep 10 seconds....\n");
        sleep(10);
        while (count < 3) {
            printf("issuing read subroutine....\n");
            n = read( fd, buf, 0x800 );
            if (n == -1) {
                return_code = errno;
                printf("read failed, errno = %d\n", errno);
                if (errno == SNA_INOP) {
                    printf("INOP received..., issue
INOP_RSP....\n");
                    n = ioctl(fd, INOP_RSP, NULL);
                    if (n == -1)
                        printf("INOP_RSP failed, errno = %d\n",
errno);
                    else
                        printf("INOP_RSP OK....\n");
                }
            }
            else {
```

```
                printf("read completed, number of bytes read =
%d\n", n);
                temp_ptr = buf;
                printf("6 bytes TH = %x %x %x %x %x %x\n",
*temp_ptr++,
                    *temp_ptr++, *temp_ptr++,*temp_ptr++, *temp_ptr+
+, *temp_ptr++);
                printf("3 bytes RH = %x %x %x\n", *temp_ptr++,
                    *temp_ptr++, *temp_ptr);
                printf("data rcvd = %s\n", &buf[9]);
            }
            buf.data[0] = 0x2c;
            buf[1] = 0x00;
            buf[2] = 0x02;
            buf[3] = 0x00;
            buf[4] = 0x00;
            buf[5] = 0x01;
            buf[6] = 0x03;
            buf[7] = 0x91;
            buf[8] = 0x01;
            temp_ptr = &buf[9];
            if (count == 0)
                strcpy(temp_ptr,
                    "This is the first reply sent by GSNALOC....");
            if (count == 1) {
                buf[2] = 0x04;
                strcpy(temp_ptr,
                    "This is the second reply sent by GSNALOC...");
            }
            if (count == 2) {
                buf[2] = 0x06;
                strcpy(temp_ptr,
                    "This is the third reply sent by GSNALOC....");
            }
            n = write( fd, buf, 132 );
            if (n == -1) {
                return_code = errno;
                printf("write failed, errno = %d\n",errno);
            }
            else
                printf("write OK, number bytes sent = %d\n",n);
            count ++;
        }
        close( fd );
    }
    exit( return_code );
}
```

Chapter 9. Sockets

The Sockets facility is a Berkeley Software Distribution (BSD) programming interface that provides applications programs with interprocess and network I/O communication capabilities. The Sockets mechanism consists of socket subroutines which enable local or remote application programs to set up virtual connections and exchange data. In AIX Version 3.1, the Sockets facility serves as the application program interface for TCP/IP. This section/chapter provides information on the Sockets facility and its components. It contains information about socket creation, connection, and use in application programs. In addition, this section/chapter provides brief descriptions of socket data structures, the socket Kernel Service Subroutines, and Network Library Subroutines.

Sockets Overview

AIX includes an implementation of the Berkeley Software Distribution (BSD) interprocess communication facility known as sockets. Sockets are communication channels that enable unrelated processes to exchange data locally and over networks. A single socket is one endpoint of the two-way communications channel. In AIX, sockets have the following characteristics:

- A socket exists only as long as some process holds a descriptor referring to it.
- Sockets are referenced by file descriptors and have qualities similar to those of a character special device. Read, write, and select operations can be performed on sockets by using the appropriate subroutines.
- Sockets may be created in pairs, or given names and used to rendezvous with other sockets in a communications domain, accepting connections from these sockets or exchanging messages with them.

Critical Attributes

Sockets share certain critical attributes which no other interprocess communication mechanisms feature. Sockets have the following attributes:

- Provide a two-way communications path.
- Have a type and one or more associated processes.
- Exist within communications domains.
- Do not require a common ancestor to set up the communication.

Application programs request the operating system to create a socket when one is needed. The operating system returns an integer that the application program uses to reference the newly created socket. Unlike file descriptors, the operating system can create sockets without binding them to a specific destination address. The application program can choose to supply a destination address each time is uses the socket.

Sockets Background

Sockets were developed in response to the need for sophisticated interprocess facilities that met the following goals:

- Provided access to communications networks such as the DARPA Internet.

- Enabled communication between unrelated processes residing locally on a single host computer and residing remotely on multiple host machines.

The development of sockets as part of the interprocess facilities was intended to provide a sufficiently general interface to allow network-based applications to be constructed independently of the underlying communications facilities and supported the construction of distributed programs built on top of communications primitives.

Note: In AIX, the socket subroutines serve as the application program interface for Transmission Control Protocol/Internet Protocol (TCP/IP).

Sockets Facilities

Socket subroutines and network library subroutines provide the building blocks for interprocess communication. An application program must perform the following basic functions to conduct interprocess communication through the socket layer:

- Creating and naming sockets.
- Accepting and making socket connections.
- Sending and receiving data.
- Shuttting down socket operations.
- Translating network addresses.

Each of these general functions is discussed briefly in this article and described in greater detail in related articles.

Creating and Naming Sockets

A socket is created with the **socket** subroutine. This subroutine creates a socket of a specified domain, type, and protocol. Sockets have different qualities depending on these specifications. A communication domain indicates the protocol families to be used with the created socket. The socket type defines its communications properties such as reliability, ordering, and prevention of duplication of messages. Some protocols families have multiple protocols that support one type of service. To supply a protocol in the creation of a socket, the programmer must understand the protocol family well enough to know the type of service each protocol supplies.

An application can bind a name to a socket. The socket names used by most applications are human-readable strings. However, the name for a socket that is used within a communication domain is usually a low-level address. The form and meaning of socket addresses are dependent on the communication domain in which the socket is created. The socket name is specified by one of three **sockaddr** structures; these structures are defined in header files.

Accepting and Making Socket Connections

Sockets may be connected or unconnected. Unconnected socketed are produced by the **socket** subroutine. An unconnected socket can yield a connected socket in one of two ways: either by actively connecting to another socket, or by becoming associated with a name in the communications domain and accepting a connection from another socket. Other types of sockets, such as datagram sockets, need not establish connections before use.

Sending and Revceiving Data

Sockets include a variety of calls for sending and receiving data. The usual **read** and **write** subroutines, as well as the socket **send** and **recv** subroutines, can be used on sockets that are in a connected state. Additional socket subroutines exists that permit callers to specify or receive the address of the peer with whom they are communicating; these calls are useful for connectionless sockets, where the peer sockets may vary on each message transmitted or received. The **sendmsg** and **recvmsg** subroutines support the full interface to the interprocess-communications facilities. Besides offering scatter-gather operations, these calls allow and address to be specified or received, and support flag options.

Shutting Down Socket Operations

Once sockets are no longer of use they can be closed or shutdown using the **shutdown** subroutine or **close** subroutine.

Translating Network Addresses

Application programs may also need to locate and construct network addresses when conducting the interprocess communication. The socket facilites include subroutines to:

- Map addresses to host names and back.
- Map network names to numbers and back.
- Extract network, host, service, and protocol names.
- Convert between varying length byte quantities.
- Resolve domain names.

Related Information

Binding Names to Sockets on page 9–13, Understanding Domain Name Resolution on page 9–24, Understanding Network Address Translation on page 9–22, Understanding Socket Addresses on page 9–8, Understanding Socket Communications Domains on page 9–6 , Understanding Socket Creation on page 9–12, Understanding Socket Connections on page 9–15, Understanding Socket Data Transfer on page 9–18, Understanding Socket Examples on page 9–26, Understanding Socket Header Files on page 9–5, Understanding the Sockets Interface on page 9–3, Understanding Socket Options on page 9–17, Understanding Socket Subroutines on page 9–5, and Understanding Socket Types and Protocols on page 9–9.

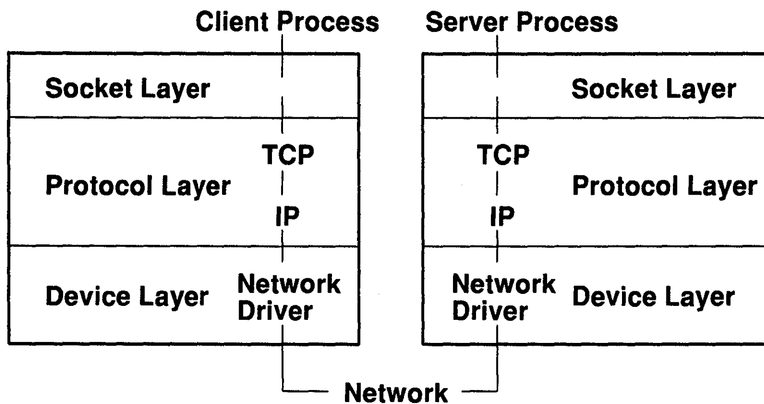
TCP/IP Overview for System Management in *Communication Concepts and Procedures.*

Networks Overview in *Communication Concepts and Procedures.*

Understanding the Sockets Interface

Sockets Interface

The kernel structure consists of three layers: the socket layer, the protocol layer, and the device layer. The socket layer supplies the interface between the subroutines and lower layers, the protocol layer contains the protocol modules used for communication, and the device layer contains the device drivers that control the network devices. In AIX Version 3.1 protocols and drivers are dynamically loadable.



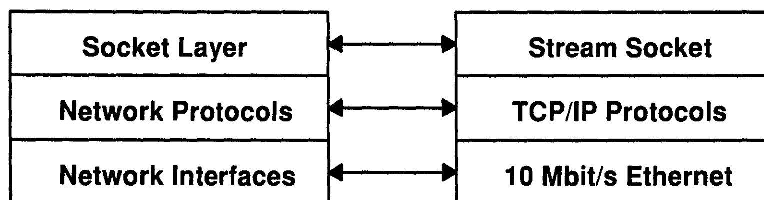
Processes communicate using the client/server model: a server process listens to a socket, one end point of a two-way communications path, and client processes communicate to the server process over another socket, the other end point of the communications path, which may be on another machine. The kernel maintains internal connections and routes data from client to server.

Within the socket layer, the socket data structure is the focus of activity. The system-call interface subroutines manage the activities related to an AIX subroutine, collecting the subroutine parameters and converting program data into the format expected by the second level subroutines.

Most of the socket facilities are implemented within the second level subroutines. These second level subroutines directly manipulate socket data structures and manage the synchronization between asynchronous activities.

Socket Interface to Network Facilities

The socket interprocess communication facilities are layered on top of networking facilities. Data flows from an application program through the socket layer to the networking support. Protocol related state is maintained in auxiliary data structures that are specific to the supporting protocols. The socket level passes responsibility for storage associated with transmitted data to the network level.



Some of the communications domains supported by the socket interprocess communications facility provide access to network protocols. These protocols are implemented as a separate software layer logically below the socket software in the kernel. The kernel provides ancillary services, such as buffer management, message routing, standardized interfaces to the protocols, and interfaces to the network interface drivers for the use of the various network protocols.

User request and control output subroutines serve as the interface from the socket subroutines to the communication protocols.

Related Information

Sockets Overview on page 9–1.

Understanding Network Interfaces for TCP/IP in *Communication Concepts and Procedures.*

Understanding Socket Subroutines

Socket Subroutines

AIX provides socket subroutines to enable interprocess and network interprocess communications. Some socket routines once contained in **(libc.a)** subroutines are now part of the AIX kernel. These routines are grouped together as the Socket Kernel Service Subroutines.

Note: Do not call any Socket Kernel Service subroutines from kernel extensions.

The socket subroutines still maintained in **(libc.a)** are grouped together under the heading of Network Library Subroutines. Application programs can use both types of socket subroutines for interprocess communication.

Related Information

Sockets Overview on page 9–1.

List of Socket Kernel Service Subroutines on page 9–27, List of Network Library Socket Subroutines on page 9–27.

Understanding Socket Header Files

Socket Header Files

Socket header files contain data definitions, structures, constants, macros, and options used by the socket subroutines and subroutines. An application program must include the appropriate header file to make use of structures or other information a particular socket subroutine or subroutine requires. Commonly used socket header files are as follows:

/usr/include/netinet.h/in.h	Defines Internet constants and structures.
/usr/include/arpa/nameser.h	Contains Internet nameserver information.
/usr/include/netdb.h	Contains data definitions for socket subroutines.
/usr/include/resolv.h	Contains resolver global definitions and variables.
/usr/include/sys/socket.h	Contains data definitions and socket structures.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains. buffer queues.
/usr/include/sys/types.h	Contains data type definitions.
/usr/include/sys/un.h	Defines structures for the UNIX Interprocess Communication domain.

In addition to the commonly used socket header files, the Internet Address Translation subroutines require the inclusion of the **inet.h** file. The **inet.h** file is located in the **/usr/include/arpa** directory.

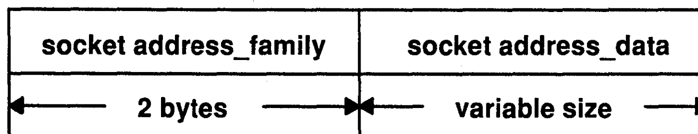
Socket Address Data structures

The socket data structure defines the socket. During a socket subroutine, the system dynamically creates the socket data structure. The socket address is specified by a data structure that is defined in a header file.

The **/usr/include/sys/socket.h** file contains the **sockaddr** structure. The **sockaddr** structure contains the following elements:

```
ushort  sa_family
char    sa_data[14];
```

The **sa_family** element contains the socket address family or domain, either **AF_UNIX** for the UNIX domain or **AF_INET** for the Internet domain. The **sa_data** element represents the socket name. The contents of the **sa_data** element depend on the protocol in use.



The types of socket address data structures are as follows:

struct sockaddr_in
Defines sockets used for machine-to-machine communication across a network and local interprocess communication. The **/usr/include/netinet/in.h** file contains the **sockaddr_in** structure. The **sockaddr_in** structure contains the following elements:

```
short    sin_family;
u_short  sin_port;
struct   in_addr sin_addr;
char     sin_zero[8};
```

struct sockaddr_un
Defines UNIX domain sockets used for local interprocess communication only. These sockets require complete path name specification and do not traverse networks. The **/usr/include/sys/un.h** file contains the **sockaddr_un** structure. The **sockaddr_un** structure contains the following elements:

```
short   sun_family;
char    sun_path[108];
```

Related Information

Sockets Overview on page 9–1.

Understanding Network Address Translation on page 9–22.

Understanding Socket Communications Domains

Sockets that share common communication properties, such as naming conventions and protocol address formats, are grouped into communications domains. A communication domain is sometimes referred to as name space or an address space.

The communication domain includes the following:

- Rules for manipulating and interpreting names
- A collection of related address formats that comprise an address family
- A set of protocols, called the protocol family.

Address formats

The address format indicates what set of rules were used in creating network addresses of a particular format. For example, in the Internet communication domain, a host address is a 32-bit value that is encoded using one of four rules based on the type of network on which the host resides.

AIX supports the UNIX and Internet communication domains. Each communication domain has different rules for valid socket names and interpretation of names. After a socket is created, it can be given a name according to the rules of the communication domain in which it was created. For example, in the UNIX communication domain sockets are named with UNIX path names (a socket can be named "/dev/foo"). Sockets normally exchange data only with sockets in the same communication domain.

Address families

The **socket** subroutine takes an address family as a parameter. Specifying an address family indicates to the system how to interpret supplied addresses. The **/usr/include/socket.h** and **/usr/include/socketvar.h** include files define the address families supported in AIX.

A socket subroutine that takes an address family as a parameter can use either **AF_UNIX** (Address Family UNIX) or **AF_INET** (Address Family Internet). These address families are part of the following communication domains:

UNIX Provides socket communication between processes running on the same AIX system when an address family of **AF_UNIX** is specified. A socket name in the UNIX domain is a string of ASCII characters whose maximum length depends on the machine in use.

Internet Provides socket communication between a local process and a process running on a remote host when an address family of **AF_INET** is specified. The Internet domain requires that TCP/IP be installed on your system. A socket name in the Internet domain is a DARPA Internet address, made up of a 32-bit IP address and a 16-bit port address.

Communications domains are defined by a loadable data structure that is defined within the operating system based on the system's configuration.

UNIX Domain Properties

Characteristics of the UNIX domain are as follows:

Types of sockets In the UNIX domain, the **SOCK_STREAM** type provides pipe-like facilities, while **SOCK_DGRAM** usually provides reliable message-style communications.

Naming Socket names are strings and appear in the UNIX file system name space through portals.

Internet Domain Properties

Characteristics of the Internet domain are as follows:

- Socket types and protocols

  SOCK_STREAM is supported by the Internet TCP protocol; **SOCK_DGRAM** by the UDP protocol. Each is layered atop the transport-level Internet Protocol (IP). The Internet Control Message Protocol is implemented atop/beside IP and is accessible through a raw socket.

- Naming

  Sockets in the Internet domain have names composed of the 32 bit Internet address, and a 16 bit port number. Options may be used to provide IP source routing or security options. The 32-bit address is composed of network and host parts; the network part is variable in size and is frequency encoded. The host part may optionally be interpreted as a subnet field plus the host on a subnet; this is enabled by setting a network address mask.

- Raw access

  The Internet domain allows a program with root user authority access to the raw facilities of IP. These interfaces are modeled as **SOCK_RAW** sockets. Each raw socket is associated with one IP protocol number, and receives, all traffic for that protocol. This allows administrative and debugging functions to occur, and enables user-level implementations of special-purpose protocols such as inter-gateway routing protocols.

Communications domains are described by a domain data structure that is loadable in AIX Version 3.1. Communications protocols within a domain are described by a structure that is defined within the system for each protocol implementation configured. When a request is made to create a socket, the system uses the name of the communication domain to search linearly the list of configured domains. If the domain is found, the domain's table of supported protocols is consulted for a protocol appropriate for the type of socket being created, or for a specific protocol request. (A wildcard entry may exist for a raw domain.) Should multiple protocol entries satisfy the request, the first is selected.

Related Information

Sockets Overview on page 9–1, Understanding Socket Header Files on page 9–5, Understanding the Sockets Interface on page 9–3.

Understanding Socket Addresses

Sockets may be named with an address so that processes can connect to them. The socket layer treats an address as an opaque object. Applications supply and receive addresses as tagged, variable length byte strings. Addresses always reside in an memory buffer (**mbuf**) on entry to the socket layer. A data structure called a **sockaddr** may be used as a template for referring to the identifying tag of each socket address.

Each address family implementation includes subroutines for address family-specific operations on addresses. When addresses must be manipulated (for example, to compare them for equality) a pointer to the address (a **sockaddr** structure) is used to extract the address family tag. This tag is then used to identify the subroutine to invoke for the desired operation.

Socket Address Storage

It is common for addresses passed in by an application program to reside in **mbufs** only long enough for the socket layer to pass them to the supporting protocol for transfer into a fixed-sized address structure. This occurs, for example, when a protocol records an address in a protocol control block. The **sockaddr** structure is the common means by which the socket layer and network-support facilities exchange addresses. The size of the generic data array was chosen to be large enough to hold most addresses directly. Communications domains that support larger addresses may ignore the array size. Only the routing data structures contain fixed-sized generic **sockaddr** structures.

- The UNIX communication domain stores filesystem pathnames in **mbufs** and allows socket names as large as 108 bytes.

- The Internet communication domain uses a structure that combines a DARPA Internet address and a port number. The Internet protocols reserve space for addresses in an Internet control-block data structure, and free up **mbufs** that contain addresses after copying their contents.

Socket Addresses in TCP/IP

The AIX TCP/IP provides a set of 16-bit port numbers within each host. Because each host assigns port numbers independently, it is possible for ports on different hosts to have the same port number. AIX TCP/IP creates the *socket address* as an identifier that is unique throughout all Internet networks. AIX TCP/IP concatenates the Internet address of the local host interface with the port number to devise the Internet socket address.

A socket is an abstract mechanism which provides an endpoint for communication. With TCP/IP, sockets are not tied to a destination address; applications sending messages can specify a different destination address for each datagram, if necessary, or they can tie the socket to a specific destination address for the duration of the connection.

Since the Internet address is always unique to a particular host on a network, the socket address for a particular socket on a particular host is unique. Additionally, since each connection is fully specified by the pair of sockets it joins, every connection between Internet hosts is also uniquely identified.

The port numbers up to 255 are reserved for official Internet services. Port numbers in the range of 256 to1023 are reserved for other well-known services that are common on Internet networks. When a client process needs one of these well-known services at a particular host, the client process sends a *service request* to the socket address for the well-known port at the host.

If a process on the host is listening at the well-known port, the server process either services the request using the well-known port or transfers the connection to another port that is temporarily assigned for the duration of the connection to the client. Using temporarily assigned (or secondary) ports frees the well-known port and allows the host well-known port to handle additional requests concurrently.

The port numbers for well-known ports are listed in the **/etc/services** file. The port numbers above 1023 are generally used by processes that need a temporary port once an initial service request has been received. These port numbers are generated randomly and used on a first-come, first-served basis.

Related Information

Sockets Overview on page 9–1, Binding Names to Sockets on page 9–13.

TCP/IP Overview for System Management in *Communication Concepts and Procedures*.

Understanding Socket Types and Protocols

In addition to a address family parameter, the socket creation subroutines take as parameters a socket type and a socket protocol. An application program specifying a socket type indicates what communication style is desired for that socket or pair of sockets. Providing a socket protocol, allows an application program to indicate a specific type of service that is desired from within allowable services in a protocol family.

Socket Types

Sockets are typed according to their communication properties. Processes usually communicate only between sockets of the same type. However, if the underlying communication protocols support the communication, communication can occur between sockets of different types.

Each socket has an associated type, which describes the semantics of communications using that socket. The socket type determines the socket communication properties such as reliability, ordering, and prevention of duplication of messages. The basic set of socket types is defined in **<sys/socket.h>**:

```
/*Standard socket types */

#define  SOCK_DGRAM          1 /*datagram*/

#define  SOCK_STREAM         2 /*virtual circuit*/

#define  SOCK_RAW            3 /*raw socket*/
```

Other socket types can be defined.

AIX supports the following types of sockets:

SOCK_DGRAM

Provides datagrams, which are connectionless messages of a fixed maximum length. This type of socket is generally used for short messages, such as a name server or time server, since the order and reliability of message delivery is not guaranteed.

In the UNIX domain, **SOCK_DGRAM** is similar to a message queue. In the Internet domain, **SOCK_DGRAM** is implemented on the UDP/IP protocol.

A *datagram* socket supports bidirectional flow of data, which is not sequenced, reliable, or unduplicated. A process receiving messages on a datagram socket may find messages duplicated, or in an order different from the sent order. Record boundaries in data are, however, preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks.

SOCK_STREAM

Provides sequenced, two-way byte streams with a transmission mechanism for stream data. This socket type transmits data on a reliable basis, in order, and with out-of-band capabilties.

In the UNIX domain, **SOCK_STREAM** works like a pipe. In the Internet domain, **SOCK_STREAM** is implemented on the TCP/IP protocol.

A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to pipes.

SOCK_RAW
Provides access to internal network protocols and interfaces. This type of socket is only available to individuals with root user authority. A raw socket allows an application to have direct access to lower-level communications protocols. Raw sockets are intended for advanced users who wish to take advantage of some protocol feature not directly accessible through a normal interface, or who wish to build new protocols atop existing low-level protocols.

Raw sockets are normally datagram-oriented, though their exact characteristics are dependent upon the interface provided by the protocol.

AIX **SOCK_DGRAM** and **SOCK_RAW** sockets allow an application program to send datagrams to correspondents named in **send** subroutines. Application programs can also receive datagrams through sockets by using the **recv** subroutines. When using **SOCK_RAW** to communicate with low-level protocols or hardware interfaces, the *Protocol* parameter is important; the application program must specify the address family in which the communication takes place.

AIX **SOCK_STREAM** sockets are full-duplex byte streams. A stream socket must be connected before any data can be sent or received on it. When using a *stream* socket for data transfer, an application program needs to perform the following sequence:

1. Create a connection to another socket with the **connect** subroutine.
2. Issue the **read** and **write** subroutines, or the **send** and **recv** subroutines to transfer data.
3. Issue the **close** subroutine to finish the session.

An application program can use the **send** and **recv** subroutines to manage out-of-band data.

SOCK_STREAM communications protocols are designed to prevent the loss or duplication of data. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable period of time, the connection is broken. When this occurs, the **socket** subroutine indicates an error with a return value of –1 and with **ETIMEDOUT** as the specific code written to the **errno** global variable. If a process sends on a broken stream, a **SIGPIPE** signal is raised. Processes that cannot handle the signal terminate. When out-of-band data arrives on a socket, a **SIGURG** signal is sent to the process group.

The process group associated with a socket can be read or set by either the **SIOCGPGRP** or the **SIOCSPGRP ioctl** operation. To receive a signal on any data, use both the **SIOCSPGRP** and **FIOASYNC ioctl** operations. These **ioctl** operations are defined in the **/sys/ioctl.h** file.

Socket Protocols

A protocol is a standard set of rules for transferring data, such as UDP/IP and TCP/IP. An application program may specify a protocol only if more than one protocol is supported for this particular socket type in this domain.

Each socket may have a specific protocol associated with it. This protocol is used within the domain to provide the semantics required by the socket type. Not all socket types are supported by each domain; support depends on the existence and implementation of a suitable protocol within the domain.

The **/usr/include/sys/socket.h** file contains a list of socket protocol families supported by AIX. The following list provides examples of protocol families found in the **socket** header file:

PF_UNIX local communication.
PF_INET DARPA Internet (TCP/IP).

These protocols are defined to be the same as their corresponding address families in the **socket** header file. Before specifing a protocol family, the programmer should check the **socket** header file for currently supported protocol families. Each protocol family consists of a set of protocols. Major protocols in the suite of Internet Network Protocols include:

- Transmisison Control Protocol (TCP)
- User Datagram Protocol (UDP)
- Internet Protocol (IP)
- Internet Control Message Protocol (ICMP).

Related Information

Sockets Overview on page 9–1, Understanding Socket Header Files on page 9–5, Understanding the Sockets Interface on page 9–3.

TCP/IP Overview for System Management, Understanding Protocols for TCP/IP in *Communication Concepts and Procedures.*

Understanding Socket Creation

Socket Creation

The basis for communication between processes centers on the socket mechanism. The socket is comparable to the AIX file access mechanism that provides an endpoint for communication. Application programs request the operating system to create a socket when one is needed through the use of socket subroutines. Subroutines used to create sockets are as follows:

- **socket**
- **socketpair**

When an application program requests the creation of a new socket, the operating system returns an integer that the application program uses to reference the newly created socket. The socket descriptor is an unsigned integer that is the lowest unused number usable for a descriptor. The descriptor is in index into the kernel descriptor table. A process can obtain a socket descriptor table by creating a socket or inheriting one from a parent process.

To create a socket with the **socket** subroutine, the application program must include a communication domain and a socket type, and it also may include a specific communication protocol within the specified communication domain.

For additional information that you may need before creating sockets, read the following concepts:

- Understanding Socket Header Files
- Understanding Socket Connections.

Related Information

The **socketpair** subroutine.

Sockets Overview on page 9–1, Understanding the Sockets Interface on page 9–3.

Networks Overview in *Communication Concepts and Procedures.*

Binding Names to Sockets

The **socket** subroutine creates a socket without a name. An unnamed socket is one without any association to local or destination addresses. Until a name is bound to a socket, processes have no way to reference it and consequently, no message can be received on it.

Communicating processes are bound by an association. The **bind** subroutine allows a process to specify half of an association: `<local address, local port>` or `<local pathname>`, while the **connect** and **accept** subroutines are used to complete a socket's association. Each domain association may have a different composite of addresses. The following domain associations are:

Internet domain Produces an association composed of local and foreign addresses, and local and foreign ports.

UNIX domain Produces an association is composed of local and foreign path names.

An application program may not care about the local address it uses and can allow the protocol software to select one. This is not true for server processes. Server processes that operate at a well-known port need to be able to specify that port to the system.

In most domains, associations must be unique. Internet domain associations must never be duplicate `<protocol, local address, local port, foreign address, foreign port>` tuples.

UNIX domain sockets need not always be bound to a name, but when bound there may never be duplicate `<protocol, local pathname, foreign pathname>` tuples. The path names may not refer to files already existing on the system.

The **bind** subroutine requires *Socket*, *Name*, and *NameLength* parameters. The socket is the integer descriptor of the socket to be bound. The *Name* parameter specifies the local address, and the *NameLength* parameter indicates the length of address in bytes. The local address is defined by a data structure generally termed **sockaddr**. The **sockaddr** structure starts with a 2-byte field that identifies the address family and is followed by information specific to that family.

In the Internet domain, a process does not have to bind an address and port number to a socket, because the **connect** and **send** subroutines automatically bind an appropriate address if they are used with an unbound socket.

The bound name is a variable-length byte string which is interpreted by the supporting protocol or protocols. Its interpretation may vary from communication domain to communication domain (this is one of the properties that comprise the *domain*). In the Internet domain a name contains an Internet address and port number. In the UNIX domain, names contain a path name and family, which is always AF_UNIX.

Binding Addresses to Sockets

Binding addresses to sockets in the Internet domain a number of considerations. Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system. Since the association is created in two steps, the association uniqueness requirement indicated previously could be violated unless care is taken. Further, user programs do not always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

Wildcard addressing is provided to aid local address binding in the Internet domain. When an address is specified as INADDR_ANY (a manifest constant defined in the **<netinet/in.h>** header file), the system interprets the address as any valid address.

Sockets with wildcard local addresses may receive messages directed to the specified port number, and sent to any of the possible addresses assigned to a host. If a server process wished to only allow hosts on a given network to connect to it, it would bind the address of the host on the appropriate network.

A local port can be specified or left as unspecified (specified as zero), in which case the system selects an appropriate port number for it.

The restriction on allocating ports was done to allow processes executing in a secure environment to perform authentication based on the originating address and port number. For example, the **rlogin(1)** command allows users to log in across a network without being asked for a password, if two conditions hold:

1. The name of the system the user is logging in from is located in the **/etc/hosts.equiv** file on the system that the user is trying to log in to (or the system name and the user name are in the user's **.rhosts** file in the user's home directory).
2. The user's **rlogin** process is coming from a privileged port on the machine from which the user is logging.

The port number and network address of the machine from which the user is logging in can be determined either by the *From* result of the **accept** subroutine, or from the **getpeername** subroutine.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application program. This is because associations are created in a two step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket still existed. To override that default port selection algorithm, a **setsockopt** subroutine must be performed prior to address binding.

The **socket** subroutine creates a socket without any association to local or destination addresses. For the Internet protocols, this means no local protocol port number has been assigned. In many cases, application programs do not care about the local address they use and are willing to allow the protocol software to choose one for them. However, server processes that operate at a well-known port must be able to specify that port to the system. Once a socket has been created, a server uses the **bind** subroutine to establish a local address for it.

Not all possible binding are valid. For example the caller might request a local protocol port that is already in use by another program, or it might request an invalid local Internet address. In such cases, the **bind** subroutine fails and returns and error message.

Obtaining Socket Addresses

New sockets sometimes inherit the set of open sockets that created them. The sockets program interface includes subroutines that allow an application to obtain the address of the destination to which a socket connects and the local address of a socket. The socket subroutines that allow a program to retrieve socket addresses are the following:

- **getsockname**
- **getpeername**

For additional information that you may need before binding or obtaining socket addresses, read the following concepts:

- Understanding Socket Header Files
- Understanding Socket Addresses
- Understanding Socket Connections.

Related Information

The **accept** subroutine, **bind** subroutine, **connect** subroutine, **getpeername** subroutine, **getsockopt** subroutine, **rlogin** command, **rhosts** file, **send** subroutine, **socket** subroutine, **socketpair** subroutine.

Sockets Overview on page 9–1, Understanding the Sockets Interface on page 9–3.

Understanding Addresses for TCP/IP in *Communication Concepts and Procedures.*

Understanding Socket Connections

Socket Connection

Initially, a socket is created in the unconnected state, which means that the socket is not associated with any foreign destination. The **connect** subroutine binds a permanent destination to a socket, placing it in the connected state. An application program must call the **connect** subroutine to establish a connection before it can transfer data through a reliable *stream* socket. Sockets used with connectionless datagram services need not be connected before they are used, but connecting sockets makes it possible to transfer data without specifying the destination each time.

The semantics of the **connect** subroutine depend on the underlying protocols. An application program desiring reliable stream delivery service in the Internet family should select the Transmission Control Protocol (TCP). In such cases, the **connect** subroutine builds a TCP connection with the destination and returns an error if it cannot. In the case of connectionless services, the **connect** subroutine does nothing more than store the destination address locally.

Connections are established between a *client* process and a *server* process. In a connection-oriented network environment, a client process initiates a connection and a server process receives, or responds to, a connection. The client and server interactions occur as follows:

- The server, when willing to offer its advertised services, binds a socket to a well-known address associated with the service, and then passively *listens* on its socket. It is then possible for an unrelated process to rendezvous with the server.
- The server process socket is marked to indicate incoming connections are to be accepted on it.
- The client requests services from the server by initiating a *connection* to the server's socket. The client process uses a **connect** subroutine to initiates a socket connection.
- If the client process's socket is unbound at the time of the **connect** call, the system automatically selects and bind a name to the socket if necessary. This is the usual way that local addresses are bound to a socket.
- The system returns an error if the connection fails (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin.

Server Connections

The server process creates a socket, binds it to a well-known protocol port, and waits for requests. If the server process uses a reliable stream delivery or the computing a response takes a significant amount of time, it may happen that a new request arrives before the server finishes responding to an old request. The **listen** subroutine allows server processes to prepare a socket for incoming connections. In terms of underlying protocols, the **listen** subroutine puts the socket in a passive mode ready to accept connections. When the server process invokes the **listen** subroutine, it also informs the operating system that the protocol software should enqueue multiple simultaneous requests that arrive at a socket. The **listen** subroutine includes a parameter that allows a process to specify the length of the request queue for that socket. If the queue is full when a connection request arrives, the operating system refuses the connection by discarding the request. The **listen** subroutine applies only to sockets that have selected reliable stream delivery service.

A server process uses the **socket**, **bind**, and **listen** subroutines to create a socket, bind it to a well-known protocol port, and specify a queue length for connection requests. Invoking the **bind** subroutine associates the socket with a well-known protocol port, but the socket is not connected to a specific foreign destination. The server process may specify a wild card allowing the socket to receive connection request from an arbitrary client.

Once a socket has been set up, the server process needs to wait for a connection. The server process waits for a connection by using the **accept** subroutine. A call to **accept** blocks until a connection request arrives. When a request arrives, the operating system returns the address of the client process that has placed the request. The operating system also creates a new socket that has its destination connected to the requesting client process and returns the new socket descriptor to the calling server process. The original socket still has a wildcard foreign destination and it still remains open.

When a connection arrives, the call to **accept** returns. The server process can either handle requests iteratively or concurrently. In the iterative approach, the server handles the request itself, closes the new socket, and then invokes the **accept** subroutine to obtain the next connection request. In the concurrent approach, after the call to **accept** returns, the server process forks a new process to handle the request. The new process inherits a copy of the new socket, so it proceeds to service the request and then exists. The original server process must close its copy of the new socket and then invoke the **accept** subroutine to obtain the next connection request.

The concurrent design for server processes results in multiple processes using the same local protocol port number. In TCP style communication, a pair of endpoints define a connection. Thus, it does not matter how many processes use a given local protocol port number as long as they connect to different destinations. In the case of a concurrent server, there is one process per client and one additional process that accepts connections. The main server process has a wildcard for the destination, allowing it to connect with an arbitrary foreign site. Each remaining process has a specific foreign destination. When a TCP data segment arrives, it is sent to the socket connected to the segment's source. If no such socket exists, the segment is sent to the socket that has a wildcard for its foreign destination. Furthermore, because the socket with a wildcard foreign destination does not have an open connection, it only honors TCP segments that request a new connection.

Connectionless Datagram Services

AIX also provides support for connectionless interactions typical of the datagram facilities found in packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

An application program can create datagram sockets using the subroutine. If a particular local address is needed, a subroutine must precede the first data transmission. Otherwise, the operating system sets the local address and/or port when data is first sent. The application program can use the subroutine and subroutines to transmit data; these calls include parameters that allow the client process to specify the address of the intended recipient of the data.

In addition to the **sendto** and **recvfrom** calls, datagram sockets can also use the **connect** subroutine to associate a socket with a specific destination address. In this case, any data sent on the socket is automatically addressed to the connected peer socket, and only data received from that peer is delivered to the client process. Only one connected address is permitted for each socket at one time; a second **connect** subroutine changes the destination address.

A **connect** subroutine requests on datagram sockets return immediately, as this simply results in the operating system recording the peer socket's address (as compared to a stream socket, where a connect request initiates establishment of an end to end connection). The **accept** and **listen** subroutines are not used with datagram sockets.

While a datagram socket is connected, errors from recent **send** subroutines may be returned asynchronously. These errors may be reported on subsequent operations on the socket, or a special socket option, SO_ERROR, used with the **getsockopt** subroutine can be used to interrogate the error status. A **select** subroutine for reading or writing returns true when a process receives an error indications. The next operation returns the error, and the error status is cleared.

For additional information that you may need before connecting sockets, read the following concepts:

- Understanding Socket Header Files
- Understanding Socket Types and Protocols.

Related Information

The **accept** subroutine, **bind** subroutine, **connect** subroutine, **listen** subroutine, **getsockopt** subroutine, **recvfrom** subroutine, **select** subroutine, **send** subroutine, **sendto** subroutine.

Sockets Overview on page 9–1, Understanding the Sockets Interface on page 9–3.

TCP/IP Overview for System Management, Understanding Protocols for TCP/IP in *Communication Concepts and Procedures.*

Understanding Socket Options

Socket Options

In addition to binding a socket to a local address or connecting it to a destination address, application programs need a method to control the socket. For example, when using protocols that use time-out and retransmission, the application program may want to obtain or set the time-out parameters. An application program may also want to control the allocation of buffer space, determine if the socket allows transmission of broadcast, or control processing of out-of-band data. The **ioctl**-style **getsockopt** and **setsockopt** subroutines provide the application program with the means to control socket operations. The **getsockopt** subroutine allows an application program to request information about socket options. The **setsockopt** subroutine allows an application program to set a socket option using the same set of values obtained with the **getsockopt** subroutine. Not all socket

options apply to all sockets. The options that can be set depend on the current state of the socket and the underlying protocol being used.

For additional information that you may need when obtaining or setting socket options, read the following concepts:

- Understanding Socket Header Files
- Understanding Socket Types and Protocols
- Out-of-Band Data

Related Information

The **getsockopt** subroutine, **setsockopt** subroutine.

Sockets Overview on page 9–1, Understanding the Sockets Interface on page 9–3.

Understanding Socket Data Transfer

Socket Data Transfer

Most of the work performed by the socket layer is in sending and receiving data. The socket layer itself explicitly refrains from imposing any structure on data transmitted or received through sockets. Any data interpretation or structuring is logically isolated in the implementation of the communication domain.

Once a connection is established between sockets, an application program can send and receive data. Sending and receiving data can be done with any one of several subroutines. The subroutines vary according to the amount of information to be transmitted and received and the state of the socket being used to perform the operation.

- The **write** subroutine may be used with a socket that is in a connected state, as the destination of the data is implicitly specified by the connection.
- The **sendto** or **sendmsg** subroutines allow the process to specify the destination for a message explicitly.
- The **read** subroutine allows a process to receive data on a connected socket without receiving the sender's address.
- The **recvfrom** and **recvmsg** subroutines allow the process to retrieve the incoming message and the sender's address.

Internally, all transmission and reception requests are converted to a uniform format and are passed to the socket-layer **sendit()** and **recvit ()** subroutines.

While the **send** subroutine and **recv** subroutine are virtually identical to the **read** and **write** subroutines, the extra flags argument in the **send** and **recv** subroutines is important. The flags, defined in the **sys/socket.h** header file, can be defined as a nonzero value if the application program requires one or more of the following:

MSG_OOB	Send/receive out-of-band data
MSG_PEEK	Look at data without reading
MSG_DONTROUTE	Send data without routing packets

Out-of-band data is specific to stream sockets. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however of general interest. When MSG_PEEK is specified with a **recv** subroutine, any data

present is returned to the user, but treated as still unread. That is, the next **read** or **recv** subroutine applied to the socket returns the data previously previewed.

Out–of–Band Data

The stream socket abstraction includes the concept of out-of-band data. Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data can be delivered to the socket independently of the normal receive queue or within the receive queue depending upon the status of SO_OOBINLINE. The abstraction defines that the out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message must contain at least one byte of data, and at least one message may be pending delivery to the user at any one time.

For communications protocols which support only in-band signaling (i.e the urgent data is delivered in sequence with the normal data), the operating system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data.

It is possible to peek at out-of-band data. If the socket has a process group, a SIGURG signal is generated when the protocol is notified of out-of-band data. A process can set the process group or process id to be informed by the SIGURG signal through a SIOCSPGRP **ioctl** call.

Note: The **/usr/include/sys/ioctl.h** file contains the **ioctl** definitions and structures for use with socket **ioctl** calls.

If multiple sockets have out-of-band data awaiting delivery, an application program can use a **select** subroutine for exceptional conditions to determine those sockets with such data pending. Neither the signal nor the select indicates the actual arrival of the out-of-band data, but only notification that is pending.

In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out-of-band data was sent. When a signal flushes any pending output, all data up to the mark in the data stream is discarded.

To send an out-of-band message the MSG_OOB flag is supplied to a **send** or a **sendto** subroutine. To receive out-of-band data, an application program must set the MSG_OOB flag when performing a **recvfrom** or **recv** subroutine.

An application program can determine if the read pointer is currently pointing at the logical mark in the data steam, by using the SIOCATMARK **ioctl** call.

A process can also read or peek at the out-of-band data without first reading up to the logical mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (e.g., the TCP protocol used to implement streams in the Internet domain). With such protocols, the out-of-band byte may not yet have arrived when a **recv** subroutine is done with the MSG_OOB flag. In that case, the call will return an error of **EWOULDBLOCK**. There may be enough in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data that the urgent data may be delivered.

Certain program that use multiple bytes of urgent data and must handle multiple urgent signals need to retain the position of urgent data within the stream. The socket-level option, SO_OOINLINE provides the capability. With this option, the position of the urgent data (the logical mark) is retained. The urgent data immediately follows the mark within the normal data stream that is returned without the MSG_OOB flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data are lost.

Socket I/O Modes

Sockets can be set to either blocking or nonblocking I/O mode. The FIONBIO **ioctl** operation is used to determine this mode. When the FIONBIO **ioctl** is set, the socket is marked nonblocking. If a read is tried and the desired data is not available, the socket does not wait for the data to become available, but returns immediately with the **EWOULDBLOCK** error code.

Note: The **EWOULDBLOCK** error code is defined with **_BSD** define and is equivalent to **EAGAIN** error code.

When the FIONBIO **ioctl** is not set, the socket is in blocking mode. In this mode, if a read is tried and the desired data is not available, the desired data is not available, the calling process waits for the data. Similarly, when writing, if **FIONBIO** is set and the output queue is full, an attempt to write causes the process to return immediately with an error code of **EWOULDBLOCK**.

When performing non-blocking I/O on sockets, a program must check for the error **EWOULDBLOCK** (stored in the global value **errno**), which occurs when an operation would normally block, but the socket it was performed on is marked as non-blocking. The following socket subroutines all return **EWOULDBLOCK**:

- **accept** subroutine
- **connect** subroutine
- **send** subroutine
- **recv** subroutine
- **read** subroutine
- **write** subroutine.

Processes using these subroutines should be prepared to deal with the **EWOULDBLOCK** return codes. If an operation such as a **send** operation cannot be done in its entirety, but partial writes are permissible (for example when using a stream socket), the data that can be sent immediately will be processed, and the return value will indicate the amount actually sent.

Related Information

Sockets Overview on page 9–1, Understanding Socket Header Files on page 9–5, Understanding the Sockets Interface on page 9–3.

Understanding Socket Shutdown

Socket Shutdown

Once a socket is no longer required, the calling program can discard the socket by applying a **close** subroutine to the socket descriptor. If a reliable delivery socket has data associated with it when a close takes place, the system continues to attempt data transfer. However, if the data is still undelivered, the system discards the data. Should the application program have no use for any pending data, it can use the **shutdown** subroutine on the socket prior to closing it.

Closing Sockets

Closing a socket and reclaiming its resouces is not always a straightforward operation. In certain situations, such as when a process exits, a **close** subroutine is never expected to fail. However, when a socket promising reliable delivery of data is closed with data still queued for transmission or awaiting acknowledgment of reception, the socket must attempt to transmit the data. If the socket discards the queued data to allow the **close** subroutine to complete successfully, it violates its promise to deliver data reliably. Discarding data can cause naive processes, which depend upon the implicit semantics of the **close** call, to work unreliably in a network environment. However, if sockets block until all data have been transmitted successfully, in some communication domains a **close** subroutine may never complete.

The socket layer compromises in an effort to address this problem yet to maintain the semantics of the **close** subroutine. In normal operation, closing a socket causes any queued but unaccepted connections to be discarded. If the socket is in a connected state, a disconnect is initiated. The socket is marked to indicate that a file descriptor is no longer referencing it, and the close operation returns successfully. When the disconnect request completes, the network support notifies the socket layer, and the socket resources are reclaimed. The network layer may attempt to transmit any data queued in the socket's send buffer, although this is not guaranteed.

Alternatively, a socket may be marked explicitly to force the application program to **linger** when closing until pending data are flushed and the connection has shutdown. This option is marked in the socket data structure using the **setsockopt** subroutine with the SO_LINGER option. The **setsockopt** subroutine, using the linger option, takes a **linger** structure. When an application program indicates that a socket is to linger, it also specifies a duration for the lingering period. If the lingering period expires before the disconnect is completed, the socket layer forcibly shuts down the socket, discarding any data still pending.

Related Information

The **close** subroutine, **linger** structure, **setsockopt** subroutine, **shutdown** subroutine,.

Sockets Overview on page 9–1.

Understanding Network Address Translation

Network Address Translation

Network Library subroutines enable an application program to locate and construct network addresses while using interprocess communications facilities in a distributed environment.

Locating a service on a remote host requires many levels of mapping before client and server can communicate. A network service is assigned a name which is intended to be understandable for a user; such as "the login server on host prospero." This name and the name of the peer host must then be translated into network addresses. Finally, the address must then be used in determining a physical location and route to the service.

Network Library subroutines map:

- Host names to network addresses.
- Network names to network numbers.
- Protocol names to protocol numbers.
- Service names to port numbers.

Additional network library subroutines exist to simplify manipulation of names and addresses.

An application program must include the **<netdb.h>** header file when using any of the Network Library subroutines.

Host Names

The following related network library subroutines map Internet host names to addresses:

- **gethostbyaddr** subroutine
- **gethostbyname** subroutine
- **sethostent** subroutine
- **endhostent** subroutine.

The official name of the host and its public aliases are returned by the **gethostbyaddr** subroutine and the **gethostbyname** subroutine, along with the address family and a null terminated list of variable length addresses. The list of variable length addresses is required because it is possible for a host to have many addresses, all with the same name.

The database for these calls is provided either by the **/etc/hosts** file or by use of a **named** nameserver. Because of the differences in the databases and their access protocols, the information returned may differ. When using the host table version of **gethostbyname** only one address is returned, but all listed aliases are included. The nameserver version may return alternate addresses but does not provide any aliases other than one given as a parameter value.

Network Names

The following related network library subroutines map network names to numbers and network numbers to names:

- **getnetbyaddr** subroutine
- **getnetbyname** subroutine
- **getnetent** subroutine
- **setnetent** subroutine
- **endnetent** subroutine.

The **getnetbyaddr** subroutine, **getnetbyname** subroutine, and **getnetent** subroutines extract their information from the **/etc/networks** file.

Protocol Names

Related network library subroutines used to map protocol names are as follows:

- **getprotobynumber** subroutine
- **getprotobyname** subroutine
- **getprotoent** subroutine
- **setprotoent** subroutine
- **endprotoent** subroutine.

The **getprotobynumber** subroutine, **getprotobyname** subroutine, and **getprotoent** subroutines extract their information from the **/etc/protocols** file.

Service Names

The following related network library subroutines map service names to port numbers:

- **getservbyname** subroutine
- **getservbyport** subroutine
- **getservent** subroutine
- **setservent** subroutine
- **endservent** subroutine.

A service is expected to reside at a specific *port* and employ a particular communication protocol. The expectation is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports. If a service resides on multiple ports, the higher level library subroutines must be bypassed or extended. Services available are contained in the **/etc/services** file.

Network Byte Order Translation

The following related network library subroutines convert network address byte order:

- **htonl** subroutine
- **htons** subroutine
- **ntohl** subroutine
- **ntohs** subroutine.

Internet Address Translation

The following related network library subroutines convert Internet addresses and dotted decimal notation:

- **inet_addr** subroutine
- **inet_lnaof** subroutine
- **inet_makeaddr** subroutine
- **inet_netof** subroutine
- **inet_network** subroutine
- **inet_ntoa** subroutine.

Network Host and Domain Names

AIX maintains an integer, called a *hostid* that identifies the host machine. Host ids fall under the category of Internet Network Addressing because, by convention, the 32-bit Internet address is used. The socket subroutines that manage the host ID are the following:

- **gethostid**
- **sethostid**

The following socket subroutines manage the internal host name are the following;

- **gethostname**
- **sethostname**

AIX maintains a string that specifies the naming domain under which the machine falls. When a site obtains authority for part of the domain name space, it invents a string that identifies its piece of the space and uses that string as the name of the domain. To manage the domain name, applications can use the following socket subroutines:

- **getdomainname**
- **setdomainname**

Related Information

Sockets Overview on page 9–1, Understanding Socket Header Files on page 9–5, Understanding the Sockets Interface on page 9–3.

Understanding Domain Name Resolution

Domain Name Resolution

When a process receives a symbolic name and needs to resolve it into an address, it calls a **resolver** subroutine. The method used by the subroutine to resolve names depends on the local host configuration. In addition, the organization of the network determines how **resolver** subroutines communicate with remote **nameserver** hosts (the hosts that resolve names for other hosts).

The **resolver** subroutines determine which type of network they are dealing with by determining whether the **/etc/resolv.conf** file exists. If the file exists, the subroutines assume that the local network has a nameserver. Otherwise, they assume that no nameserver is present.

To resolve a name with no nameserver present, the **resolver** subroutine checks the **/etc/hosts** file file for an entry that maps the name to an address.

To resolve a name in a nameserver network, the **resolver** subroutine first queries the domain nameserver database, which may be local if the host is a domain name server or may be on a foreign host. If the subroutine is using a remote name server, the subroutine uses the Domain Name Protocol (DOMAIN) to query for the mapping. If this query fails, the subroutine then checks for an entry in the local **/etc/hosts** file.

Resolver subroutines are used to make, send, and interpret packets for name servers in the Internet domain. Together the following subroutines form the **resolver**, a set of functions that resolves domain names:

- **res_mkquery** subroutine
- **res_send** subroutine
- **res_init** subroutine

- **dn_comp** subroutine
- **dn_expand** subroutine
- **dn_skip** subroutine
- **dn_find** subroutine
- **getshort** subroutine
- **getlong** subroutine
- **putshort** subroutine
- **putlong** subroutine .

Global information that is used by these resolver subroutines is kept in the **_res** structure. This structure is defined in the **/usr/include/resolv.h** header file, and it contains the following members:

int	retrans;	
int	retry;	
long	options;	
int	nscount;	
struct	sockaddr_in	nsaddr_list [MAXNS];
ushort	id;	
char	defdname [MAXDNAME];	
#define	nsaddr nsaddr_list [0]	

The **options** field of the **_res** structure is constructed by logically ORing the following values:

RES_INIT Indicates whether the initial name server and default domain name have been initialized (that is, whether the **res_init** subroutine has been called).

RES_DEBUG Prints debugging messages.

RES_USEVC Uses TCP/IP connections for queries instead of UDP/IP.

RES_STAYOPEN Used with RES_USEVC, keeps the TCP/IP connection open between queries. While UDP/IP is the mode normally used, TCP/IP mode and this **option** are useful for programs that regularly perform many queries.

RES_RECURSE Sets the Recursion Desired bit for queries. This is the default.

Note that the **res_send** subroutine does not perform iterative queries and expects the name server to handle recursion.

RES_DEFNAMES Appends the default domain name to single label queries. This is the default.

Related Information

Sockets Overview on page 9–1, Understanding Socket Header Files on page 9–5, Understanding the Sockets Interface on page 9–3.

TCP/IP Overview for System Management, Domain Name Protocol, Understanding Naming for TCP/IP in *Communication Concepts and Procedures.*

Understanding Socket Examples

Socket Examples

The socket examples are programming fragments illustrating a socket function. They cannot be used in an application program without modification. They are intended only for illustrative purposes and not for use within a program.

All socket applications must be compiled with **_BSD** defined. In addition, most should probably include the libBSD BSD librar .

Related Information

Sockets Overview on page 9–1, Understanding Socket Header Files on page 9–5.

List of Socket Examples

List of Socket Kernel Service Subroutines

accept Accepts a connection on a socket to create a new socket.

bind Binds a name to a socket.

connect Connects two sockets.

getdomainname Gets the name of the current domain.

gethostid Gets the unique identifier of the current host.

gethostname Gets the unique name of the current host.

getpeername Gets the name of the peer socket.

getsockname Gets the socket name.

getsockopt Gets options on sockets.

listen Listens for socket connections and limits the backlog of incoming connections.

recv Receives messages from connected sockets.

recvfrom Receives messages from sockets.

recvmsg Receives a message from any socket.

send Sends messages from a connected socket.

sendmsg Sends a message from a socket by using a message structure.

sendto Sends messages through a socket.

setdomainname Sets the name of the current domain.

sethostid Sets the unique identifier of the current host.

sethostname Sets the unique name of the current host.

setsockopt Sets socket options.

shutdown Shuts down all socket send and receive operations.

socket Creates an end point for communication and returns a descriptor.

socketpair Creates a pair of connected sockets.

List of Network Library Socket Subroutines

dn_comp Compresses a domain name.

dn_expand Expands a compressed domain name.

dn_find Searches for an expanded domain name.

dn_skipname Skips over a compressed domain name.

endhostent Ends retrieval of network host entries.

endnetent Closes the **networks** file.

endprotoent Closes the **/etc/protocols** file.

endservent Closes the **/etc/service** file entry.

gethostbyaddr
Gets network host entry by address.

gethostbyname
Gets network host entry by name.

gethostent Gets host entry from the **/etc/hosts** file.

getnetbyaddr Gets network entry by address.

getnetbyname Gets network entry by name.

getnetent Gets network entry.

getprotobyname
Gets protocol entry from the **/etc/protocols** file by protocol name.

getprotbynumber
Gets a protocol entry from the **/etc/protocols** file by number.

getprotoent Gets protocol entry from the **/etc/protocols** file.

_getlong Retrieves long byte quantities.

_getshort Retrieves short byte quantities.

getservbyname
Gets service entry by name.

getservbyport Gets service entry by port.

getservent Gets services file entry.

htonl Converts an unsigned long integer from host byte order to Internet network byte order.

htons Converts an unsigned short integer from host byte order to Internet network byte order.

inet_addr Converts Internet addresses to Internet numbers.

inet_lnaof Separates local Internet addresses into their network number and local network address.

inet_makeaddr
Makes an Internet address.

inet_netof Seperates network Internet addresses into their network number and local network address.

inet_network Converts Internet network addresses in . (dot) notation to Internet numbers.

inet_ntoa Converts an Internet address into an ASCII string.

ntohl Converts an unsigned long integer from Internet network standard byte order to host byte order.

ntohs Converts an unsigned short integer from Internet network byte order to host byte order.

_putlong Places long byte quantities into the byte stream.

_putshort Places short byte quantities into the byte stream.

rcmd Allows execution of commands on a remote host.

res_init	Searches for a default domain name and Internet address.
res_mkquery	Makes query messages for name server.
res_send	Sends a query to a name server and retrieves a response.
rexec	Allows command execution on a remote host.
rresvport	Retrieves a socket with a privileged address.
ruserok	Allows servers to authenticate clients.
sethostent	Opens network host file.
setnetent	Opens and rewinds the network file.
setprotoent	Opens and rewinds the /**etc**/**protocols** file.
setservent	Gets service file entry.

List of Socket Header Files

/usr/include/netinet/in.h	Defines Internet constants and structures.
/usr/include/arpa/nameser.h	Contains Internet nameserver information.
/usr/include/netdb.h	Contains data definitions for socket subroutines.
/usr/include/resolv.h	Contains resolver global definitions and variables.
/usr/include/sys/socket.h	Contains data definitions and socket structures.
/usr/include/sys/socketvar.h	Defines the kernel structure per socket and contains buffer queues.
/usr/include/sys/types.h	Contains data type definitions.
/usr/include/sys/un.h	Defines structures for the UNIX Interprocess Communication domain.

List of Socket Examples

- Socketpair Communication Example Program
- Reading UNIX Datagrams Example Program
- Sending UNIX Datagrams Example Program
- Reading Internet Datagrams Example Program
- Sending Internet Datagrams Example Programs
- Initiating Internet Stream Connections Example Program
- Accepting Internet Stream Connections Example Program
- Initiating UNIX Stream Connections Example Program
- Accepting UNIX Stream Connections Example Program
- Checking for Pending Connections Example Program.

Socketpair Communication Example Program

The following program is incomplete and is intended to only illustrate the use of the socket subroutines. It cannot be compiled, linked, or executed without modification.

```
/* This program fragment creates a pair of connected sockets then
 * forks and communicates over them.  Socketpairs have a two-way
 * communication path.  Messages can be sent in both directions.
 */

#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>

#define DATA1 "In Xanadu, did Kublai Khan..."
#define DATA2 "A stately pleasure dome decree..."

main()
{
   int sockets[2], child;
   char buf[1024];

   if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0) {
      perror("opening stream socket pair");
      exit(1);
   }

   if ((child = fork()) == -1)
      perror("fork");
   else if (child) {     /* This is the parent. */
      close(sockets[0]);
      if (read(sockets[1], buf, 1024, 0) < 0)
         perror("reading stream message");
      printf("-->%s\n", buf);
      if (write(sockets[1], DATA2, sizeof(DATA2)) < 0)
         perror("writing stream message");
      close(sockets[1]);

   } else {     /* This is the child. */
      close(sockets[1]);
      if (write(sockets[0], DATA1, sizeof(DATA1)) < 0)
         perror("writing stream message");
      if (read(sockets[0], buf, 1024, 0) < 0)
         perror("reading stream message");
      printf("-->%s\n", buf);
      close(sockets[0]);
   }
}
```

Reading UNIX Datagrams Example Program

The following program is incomplete and is intended to only illustrate the use of socket subroutines. It cannot be compiled, linked, or executed without modification.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

/*
 * In the include file <sys/un.h>, a sockaddr_un is defined as
 * follows:
 * struct sockaddr_un {
 *   short sun_family;
 *   char sun_path[108];
 * };
 */

#include <stdio.h>

#define NAME "socket"

/*
 * This program creates a UNIX domain datagram socket, binds a
 * name to it, then reads from the socket.
 */
main()
{
   int sock, length;
   struct sockaddr_un name;
   char buf[1024];

   /* Create socket from which to read. */
   sock = socket(AF_UNIX, SOCK_DGRAM, 0);
   if (sock < 0) {
      perror("opening datagram socket");
      exit(1);
   }

 /* Create name. */
   name.sun_family = AF_UNIX;
   strcpy(name.sun_path, NAME);
   if (bind(sock, &name, sizeof(struct sockaddr_un))) {
      perror("binding name to datagram socket");
      exit(1);
   }

 printf("socket -->%s\n", NAME);
   /* Read from the socket */
   if (read(sock, buf, 1024) < 0)
      perror("receiving datagram packet");
   printf("-->%s\n", buf);
   close(sock);
   unlink(NAME);
}
```

Sending UNIX Datagrams Example Program

The following program is incomplete and is intended to only illustrate the use of the socket subroutines. It cannot be compiled, linked, or executed without modification.

```
/*
 * This program fragment sends a datagram to a receiver whose
 * name is retrieved from the command line arguments.  The form
 * of the command line is <udgramsend pathname>.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define DATA "The sea is calm tonight, the tide is full..."

main(argc, argv)
   int argc;
   char *argv[];
{
   int sock;
   struct sockaddr_un name;

   /* Create socket on which to send. */
   sock = socket(AF_UNIX, SOCK_DGRAM, 0);
   if (sock < 0) {
      perror("opening datagram socket");
      exit(1);
   }
   /* Construct name of socket to send to. */
   name.sun_family = AF_UNIX;
   strcpy(name.sun_path, argv[1]);

  /* Send message. */
   if (sendto(sock, DATA, sizeof(DATA), 0, &name,
       sizeof(struct sockaddr_un)) < 0) {
      perror("sending datagram message");
   }
   close(sock);
}
```

Reading Internet Datagrams Example Program

The following program is incomplete and is intended to only illustrate the use of the socket subroutines. It cannot be compiled, linked, or executed without modification.

```
/*
 * In the include file <netinet/in.h>, a sockaddr_in is defined as
 * follows:
 * struct sockaddr_in {
 *    short sin_family;
 *    u_short sin_port;
 *    struct in_addr sin_addr;
 *    char sin_zero[8];
 * };
 * This program creates a datagram socket, binds a name to it,
 * then reads from the socket.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

main()
{
   int sock, length;
   struct sockaddr_in name;
   char buf[1024];

   /* Create socket from which to read. */
   sock = socket(AF_INET, SOCK_DGRAM, 0);
   if (sock < 0) {
      perror("opening datagram socket");
      exit(1);
   }
   /* Create name with wildcards. */
   name.sin_family = AF_INET;
   name.sin_addr.s_addr = INADDR_ANY;
   name.sin_port = 0;
   if (bind(sock, &name, sizeof(name))) {
      perror("binding datagram socket");
   exit(1);
   }
 /* Find assigned port value and print it out. */
   length = sizeof(name)
   if (getsockname(sock, &name, &length)) {
      perror("getting socket name");
      exit(1);
   }
 printf("Socket has port #%d\n", ntohs(name.sin_port));
   /* Read from the socket */
   if (read(sock, buf, 1024) < 0)
      perror("receiving datagram packet");
   printf("-->%s\n", buf);
   close(sock);
}
```

Sending Internet Datagrams Example Program

The following program is incomplete and is intended to only illustrate the use of the socket subroutines. It cannot be compiled, linked, or executed without modification.

```
/*
 * This program fragment sends a datagram to a receiver whose
   name is retrieved from the command line arguments.  The form
   of the command line is <dgramsend hostname portnumber>.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "The sea is calm tonight, the tide is full..."

main(argc, argv)
   int argc;
   char *argv[];
{
   int sock;
   struct sockaddr_in name;
   struct hostent *hp, *gethostbyname();

   /* Create socket on which to send. */
   sock = socket(AF_INET, SOCK_DGRAM, 0);
   if (sock < 0) {
      perror("opening datagram socket");
      exit(1);
   }


   /*
    * Construct name, with no wildcards, of the socket to send to.
    * Gethostbyname() returns a structure including the network
    * address of the specified host.  The port number is taken
    * from the command line.
    */
   hp = gethostbyname(argv[1]);
   if (hp == 0) {
      fprintf(stderr, "%s: unknown host0", argv[1]);
      exit(2);
   }
   bcopy(hp->h_addr, &name.sin_addr, hp->h_length);
   name.sin_family = AF_INET;
   name.sin_port = htons(atoi(argv[2]));
   /* Send message. */
   if (sendto(sock, DATA, sizeof(DATA), 0, &name,
       sizeof(name)) < 0)
      perror("sending datagram message");
   close(sock);
}
```

Initiating Internet Stream Connections Example Program

The following program is incomplete and is intended to only illustrate the use of the socket subroutines. It cannot be compiled, linked, or executed without modification.

```
/*
 * This program creates a socket and initiates a connection with
 * the socket given in the command line.  One message is sent over
 * the connection and then the socket is closed, ending the
 * connection.  The form of the command line is <streamwrite
 * hostname portnumber>.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "Half a league, half a league..."

main(argc, argv)
   int argc;
   char *argv[];
{
   int sock;
   struct sockaddr_in server;
   struct hostent *hp, *gethostbyname();
   char buf[1024];


   /* Create socket */
   sock = socket(AF_INET, SOCK_STREAM, 0);
   if (sock < 0) {
      perror("opening stream socket");
      exit(1);
   }
   /* Connect socket using name specified by command line. */
   server.sin_family = AF_INET;
   hp = gethostbyname(argv[1]);
   if (hp == 0) {
      fprint(stderr, "%s: unknown host0", argv[1]);
      exit(2);
   }
   bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
   server.sin_port = htons(atoi(argv[2]));



   if (connect(sock, &server, sizeof(server)) , 0) {
      perror("connecting stream socket");
      exit(1);
   }
   if (write(sock, DATA, sizeof(DATA)) , 0)
      perror("writing on stream socket");
   close(sock);
}
```

Accepting Internet Stream Connections Example Program

The following program is incomplete and is intended to only illustrate the use of the socket subroutines. It cannot be compiled, linked, or executed without modification.

```
/*
 * This program creates a socket and then begins an infinite loop.
 * Each time through the loop it accepts a connection and prints
 * out messages from it.  When the connection breaks, or a
 * termination message comes through, the program accepts a new
 * connection.
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1


main()
{
   int sock, length;
   struct sockaddr_in server;
   int msgsock;
   char buf[1024];
   int rval;
   int i;

   /* Create socket */
   sock = socket(AF_INET, SOCK_STREAM, 0);
   if (sock < 0) {
      perror("opening stream socket");
      exit(1);
   }
   /* Name socket using wildcards. */
   server.sin_family = AF_INET;
   server.sin_addr.s_addr = INADDR_ANY;
   server.sin_port = 0;
   if (bind(sock, &server, sizeof(server))) {
      perror("binding stream socket");
      exit(1);
   }
   /* Find out assigned port number and print it out */
   length = sizeof(server);
   if (getsockname(sock, &server, &length)) {
      perror("getting socket name");
      exit(1);
   }
   printf("Socket has port #%d\n", ntohs(server.sin_port));
```

```
    /* Start accepting connection */
    listen(sock, 5);
    do {
       msgsock = accept(sock, 0, 0);
       if (msgsock == -1)
          perror("accept");
       else do {
          bzero(buf, sizeof(buf));
          if ((rval = read(msgsock, buf, 1024)) < 0)
             perror("reading stream message");
          i = 0;
          if (rval == 0)
             printf("Ending connection\n");
          else
             print("-->%s\n", buf);
       } while (rval != 0);
       close(msgsock);
    } while (TRUE);
    /*
     * Since this program has an infinite loop, the socket "sock"
     * is never explicitly closed.  However, all sockets are
     * closed automatically when a process is killed or terminates
     * normally.
     */
}
```

Initiating UNIX Stream Connections Example Program

The following program is incomplete and is intended to only illustrate the use of the socket subroutines. It cannot be compiled, linked, or executed without modification.

```
/*
 * This program connects to the socket named in the command line
 * and sends a one line message to that socket.  The form of the
 * command line is <ustreamwrite pathname>.
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define DATA "Half a league, half a league..."

main(argc, argv)
   int argc;
   char *argv[];
{
   int sock;
   struct sockaddr_in server;
   char buf[1024];

   /* Create socket */
   sock = socket(AF_UNIX, SOCK_STREAM, 0);
   if (sock < 0) {
      perror("opening stream socket");
      exit(1);
   }

/* Connect socket using name specified by command line. */
   server.sun_family = AF_UNIX;
   strcpy(server.sun_path, argv[1]);

   if (connect(sock, &server, sizeof(struct sockaddr_un) < 0) {
      close(sock);
      perror("connecting stream socket");
      exit(1);
   }
   if (write(sock, DATA, sizeof(DATA)) < 0)
      perror("writing on stream socket");
}
```

Accepting UNIX Stream Connections Example Program

The following program is incomplete and is intended to only illustrate the use of the socket subroutines. It cannot be compiled, linked, or executed without modification.

```
/*
 * This program creates a socket in the UNIX domain and binds a
 * name to it. After printing the socket's name it begins a loop.
 * Each time through the loop it accepts a connection and prints
 * out messages from it. When the connection breaks, or a
 * termination message comes through, the program accepts a new
 * connection.
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define NAME "socket"

main()
{  int sock, msgsock, rval;
   struct sockaddr_un server;
   char buf[1024];

   /* Create socket */
   sock = socket(AF_UNIX, SOCK_STREAM, 0);
   if (sock < 0) {
      perror("opening stream socket");
      exit(1);
   }
   /* Name socket using file system name */
   server.sun_family = AF_UNIX;
   strcpy(server.sun_path, NAME);
   if (bind(sock, &server, sizeof(struct sockaddr_un))) {
      perror("binding stream socket");
      exit(1);
   }

printf("Socket has name %s\n", server.sun_path);
   /* Start accepting connections */
   listen(sock, 5);
   for (;;) {
```

```
      msgsock = accept(sock, 0, 0);
       if (msgsock == -1)
          perror("accept");
       else do {
          bzero(buf, sizeof(buf));
          if ((rval = read(msgsock, buf, 1024)) <0 )
             perror("reading stream message");
          else if (rval == 0)
             printf("Ending connection\n");
          else
             printf("-->%s\n", buf);
       } while (rval > 0);
       close(msgsock);
    }

    /* The following statements are not executed, because they
     * follow an infinite loop.  However, most ordinary programs
     * will not run forever.  In the UNIX domain it is necessary to
     * tell the file system that one is through using NAME.  In
     * most programs one uses the call unlink() as below.  Since
     * the user will have to kill this program,  it will be
     * necessary to remove the name by a command from the shell.
     */
    close(sock);
    unlink(NAME);
}
```

Checking for Pending Connections Example Program

The following program is incomplete and is intended to only illustrate the use of the socket subroutines. It cannot be compiled, linked, or executed without modification.

```
/*
 * This program uses select() to check that someone is trying to
 * connect before calling accept().
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

main()
{
   int sock, length;
   struct sockaddr_in server;
   int msgsock;
   char buf[1024];
   int rval;
   fd_set ready;
   struct timeval to;

   /* Create socket */
   sock = socket(AF_INET, SOCK_STREAM, 0);
   if (sock < 0) {
      perror("opening stream socket");
      exit(1);
   }
   /* Name socket using wildcards. */
   server.sin_family = AF_INET;
   server.sin_addr.s_addr = INADDR_ANY;
   server.sin_port = 0;
   if (bind(sock, &server, sizeof(server))) {
      perror("binding stream socket");
      exit(1);
   }


   /* Find out assigned port number and print it out */
   length = sizeof(server);
   if (getsockname(sock, &server, &length)) {
      perror("getting socket name");
      exit(1);
   }
   printf("Socket has port #%d\n", ntohs(server.sin_port));
```

```
/* Start accepting connections */
  listen(sock, 5);
  do {
     FD_ZERO(&ready);
     FD_SET(sock, &ready);
     to.tv_sec = 5;
     if (select(sock + 1, &ready, 0, 0, &to) < 0) {
        perror("select");
        continue;
     }
     if (FD_ISSET(sock, &ready)) {
        msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
        if (msgsock == -1)
           perror("accept");
        else do {
           bzero(buf, sizeof(buf));
           if ((rval = read(msgsock, buf, 1024)) < 0)
              perror("reading stream message");
           else if (rval == 0)
              printf("Ending connection\n");
           else
              print("-->%s\n", buf);
        } while (rval > 0);
        close(msgsock);
     } else
        printf("Do something else\n");
  } while (TRUE);
}
```

Chapter 10. X.25 Communications

X.25 is an international standard protocol that allows intercommunication between systems. It is particularly useful for communicating with people using different computer systems and for applications that access public data bases.

There are both public networks and private networks based on X.25. Public networks are usually provided on a national basis by the national Post, Telegraph and Telecommunications (PTT) authority. Private networks are operated by individual corporations, to help them carry out their business.

X.25 Overview

The use of an X.25 network is an airline reservation system; another is for an order entry system for automobile parts. Many of the corporations using X.25 networks have a requirement for communication between themselves and other companies such as dealers and agents.

You can use X.25 communications on the RISC System/6000 to provide a network service for higher-level protocols, such as SNA and TCP/IP. Or you can use an X.25 network directly, either by using the **xtalk** command, or by using the application programming interface (API) to write your own applications.

Before you can use the X.25 API, X.25 communications must be installed and configured. You must install the base operating system extension that includes the X.25 application programming interface (API). You also need access to a C compiler.

For information about the X.25 protocol, managing X.25 communications on the RISC System/6000, and using the **xtalk** command, see the *Communications Concepts and Procedures* book. The X.25 subroutines are included in the *Calls and Subroutines Reference* book; the **x25sdefs.h** header file is included in the *Files Reference* book. For an introduction to application programming for X.25, continue with the following:

- The X.25 Application Programming Interface (API)
- Using the X.25 Subroutines
- Using X.25 Applications Written For Previous Releases
- Using the X.25 Structures and Flags
- Using Processes in X.25 Applications
- Providing Security in X.25 Applications
- Understanding X.25 Error Codes

The X.25 Application Programming Interface (API)

In addition to the X.25 commands, such as **xtalk**, which can be used as soon as you have set up X.25 communications, there is an application programming interface (API), with which you can write your own applications, especially tailored to your own users' needs.

When you use the X.25 application programming interface, you are at a level slightly above the packet level. Although it helps to have an overview of what is going on underneath, you do not need to be concerned about all the details.

What the X.25 API Includes

The X.25 API includes a library of C subroutines that use the services of the X.25 adapter and adapter code. Your application programs call these subroutines when they need X.25 functions. The subroutines use a number of structures to pass information between them and the application programs. Further information is to be found in:

- Using the X.25 Subroutines
- Using the X.25 Structures and Flags.

The X.25 API provides the following types of identifiers for use in programs:

- Listen identifiers, **listen_id,** for potential incoming calls
- Connection identifiers, **conn_id,** for established calls
- Counter identifiers, **ctr_id,** for notification of incoming messages.

In addition to the subroutine library and the header files for the structures, there are also some example programs that demonstrate the use of the subroutines. Further information is to be found in:

- X.25 Example Programs Overview.

If you already have applications written using the X.25 API for a previous release of AIX, and want to port them to this release, you should look at:

- Using X.25 Applications Written For Previous Releases.

Learning How to Use the API

Background information for using the subroutines is included in:

- X.25 API: Initializing and Terminating
- X.25 API: Using the Connection Identifier for Calls
- X.25 API: Using Counters to Correlate Messages
- X.25 API: Listening for Incoming Calls
- X.25 API: Making and Receiving a Call
- X.25 API: Transferring and Acknowledging Data
- X.25 API: Clearing, Resetting, and Interrupting Calls.

Using the X.25 Subroutines

The X.25 application programming interface includes the following groups of subroutines for use in C programs:

- Initialization and termination subroutines
- Network subroutines
- Counter subroutines
- Management subroutines.

The X.25 API subroutines are kept in the **/usr/lib/libx25s.a** library. You call the API subroutines using standard C conventions.

Compile your program by using this command:

`cc` *sourcefilename* `–lx25s`

When you call one of the subroutines, the outcome can be either normal or one of a number of error conditions. All the possible X.25 error conditions are listed and explained in the List of X.25 API Error Codes and the possible error conditions for each subroutine are listed in the reference article for that subroutine. When writing applications, you should test for any possible error conditions and take appropriate action.

List of X.25 Initialization and Termination Subroutines

x25_init	Initializes the API for a particular X.25 port.
x25_term	Terminates the API for a particular X.25 port.

List of X.25 Network Subroutines

These are the subroutines that establish calls, transmit data, clear calls, and make other use of the network.

x25_ack	Acknowledges data received with the D-bit set.
x25_call	Sets up a switched virtual circuit and establishes the call.
x25_call_accept	Accepts an incoming call.
x25_call_clear	Clears a call.
x25_listen	Starts listening for incoming calls.
x25_deafen	Turns off listening.
x25_interrupt	Sends an interrupt.
x25_pvc_alloc	Allocates a permanent virtual circuit.
x25_pvc_free	Frees a permanent virtual circuit.
x25_receive	Receives a message and indicates the message type.
x25_reset	Resynchronizes communications.
x25_reset_confirm	Sends a reset-confirmation message.
x25_send	Sends data.

List of X.25 Counter Subroutines

To understand and control what is happening during a call, your application should use counters supplied by the API.

x25_ctr_get	Gets a counter.
x25_ctr_remove	Removes a counter.
x25_ctr_test	Gets the current value of a counter.
x25_ctr_wait	Suspends the current process until one of the counters has exceeded a specified value, usually zero.

List of X.25 Management Subroutines

These subroutines can be used to control and monitor X.25 links.

x25_link_connect	Connects an X.25 port to the X.25 network.
x25_link_disconnect	Disconnects an X.25 port from the X.25 network.
x25_link_query	Queries the status of the X.25 port.
x25_link_monitor	Controls monitoring of an X.25 port.
x25_link_statistics	Obtains statistics for an X.25 port.
x25_device_query	Returns information about some of the attributes of an X.25 adapter.
x25_circuit_query	Returns information about a virtual circuit.

Using X.25 Applications Written for Previous Releases

If you have applications written for AIX 2.2, you will have to make a few changes to them before you can run them on this release. You will be able to take advantage of some new functions. The changes to the API include:

- New, changed, and obsolete subroutines
- Changes to structures
- Other changes.

Changed, New, and Obsolete Subroutines

To support multiple X.25 adapters, seven of the subroutines now have a **link_name** parameter.

There are five new subroutines: **x25_term**, **x25_circuit_query**, **x25_device_query**, **x25_link_query**, and **x25_link_statistics**.

The **x25_query_device** subroutine has been replaced by **x25_circuit_query** and **x25_device_query**. The **x25_link_status** subroutine has been replaced by **x25_link_query**.

All the subroutines are listed here:

- Initialization and termination subroutines
- Network subroutines
- Counter subroutines
- Management subroutines .

Changes to Initialization and Termination Subroutines

x25_init	Now has the **link_name** parameter.
x25_term	New.

Changes to Network Subroutines

x25_ack	Unchanged.
x25_call	Unchanged.
x25_call_accept	Unchanged.
x25_call_clear	Now has the **cb_msg** parameter.
x25_deafen	Unchanged.
x25_interrupt	Unchanged.
x25_listen	Unchanged.
x25_pvc_alloc	Now has the **pvc_ptr** parameter instead of the **channel** parameter.
x25_pvc_free	Unchanged.
x25_receive	Unchanged.
x25_reset	Unchanged.
x25_reset_confirm	Unchanged.
x25_send	Unchanged.

Changes to Counter Subroutines

x25_ctr_get	Unchanged.
x25_ctr_remove	Unchanged.
x25_ctr_test	Unchanged.
x25_ctr_wait	Unchanged.

Changes to Management Subroutines

x25_circuit_query	Replaces some of functions of the **x25_query_device** subroutine.
x25_device_query	Replaces some of functions of the **x25_query_device** subroutine; it has the **link_name** parameter.
x25_link_connect	Now has the **link_name** parameter.
x25_link_disconnect	Now has the **link_name** parameter.
x25_link_query	Replaces the **x25_link_status** subroutine and now has the **link_name** parameter.
x25_link_monitor	Now has the **link_name** parameter.
x25_link_status	Has been replaced by **x25_link_query**.
x25_link_statistics	New.
x25_query_device	Has been replaced by **x25_circuit_query** and **x25_device_query**.

Changes to Structures

The structures that you include in your programs have been changed to allow for possible future developments in the X.25 protocol. Every field now has a flag associated with it, which you set to indicate that you are using the field.

Other changes to the structures are to support the 1984 version of X.25.

Other Changes

The X.25 API no longer uses shared memory.

Due to changes in interprocess communication, you no longer have to issue **setgrp()** at the beginning of your programs.

Using the X.25 Structures and Flags

Structures

With many of the subroutines, you enter the parameters into a structure, and pass to the subroutine a pointer to this structure. Definitions of these structures are supplied as a header file, **/usr/include/x25sdefs.h**. Include the following line in your programs:

```
#include <x25sdefs.h>
```

The List of X.25 API Structures lists all the structures included in the **/usr/include/x25sdefs.h** file.

Flags

Each of the fields in a structure has a flag associated with it. This flag tells the API whether the associated field has been used; if the corresponding flag has not been set, the field is ignored by the API. To use the flag, which is a constant, OR it with the **unsigned long flags** in the structure. This sets the appropriate bit in **flags**.

Before invoking a subroutine, the appropriate **flags** field must be set to 0 or to a particular flag constant. For example, to set the **flags** field to 0 before invoking **x25_call**:

```
cb_call.flags = 0
```

To indicate that the **link_name** field is being used, before invoking **x25_call**:

```
cb_call.flags = X25FLG_LINK_NAME
```

There are also some flags (for instance, **X25FLG_D_BIT**) that do not correspond to structure elements.

Understanding X.25 Error Codes

The X.25 subroutines set **x25_errno** and **errno** to indicate error conditions.

How x25_errno and errno Are Used

If an error condition results from an X.25 API subroutine call, it is indicated in one of the following ways, depending on the type of error:

- For X.25-specific error conditions, **x25_errno** indicates the error, for example `X25ACKREQ`; **errno** is not set in these conditions.
- For other error conditions, **x25_errno** is set to `X25SYSERR`; **errno** indicates the error, for example, `EFAULT`.

In a production program, you should handle each condition that is likely to occur, giving the end user a message telling them what action to take. We give you a code example of how to do this, for the **x25_link_statistics** subroutine. All the other subroutines can be handled in a similar way.

The List of X.25 API Error Codes lists the error codes that may be returned by X.25 subroutines.

Using Processes in X.25 Applications

It is a good idea to divide applications into multiple processes. For instance, it is useful to fork off a separate process for sending or receiving data, after the call has been established. Or you could have one process to listen for calls and one process to make calls. Use of multiple processes may improve performance.

Child processes should not be created while a call is being established; that is to say, you could fork off a process:

- Before making a call
- After receiving the call confirmation
- Before receiving an incoming call.

A connection identifier can be used by the process that made or received the call, or its children; it cannot be used by other processes.

A child process can use a virtual circuit established by its parent, but a parent process cannot use a virtual circuit established by its child.

Providing Security in X.25 Applications

Security mechanisms are built into the hardware and the operating system. In addition, the use of the management subroutines is restricted to users as follows:

Security Permissions Needed for the X.25 Management Subroutines

x25_circuit_query	No permission needed.
x25_device_query	No permission needed.
x25_link_query	No permission needed.
x25_link_statistics	No permission needed.
x25_link_connect	NET_CONFIG permission.
x25_link_disconnect	NET_CONFIG permission.
x25_link_monitor	NET_CONFIG and RAS_CONFIG permission.

The various identifiers used by the API also have restrictions:

See Restrictions on the Use of the Connection Identifer on page 10–10.
See Restrictions on the Use of Counters on page 10–12.
See Restrictions on the Use of the Listen Identifier on page 10–12.

Related Information

x25sdefs.h File.

Header Files Overview, Subroutines Overview and Compiling, Linking, and Running Programs in *General Programming Concepts.*

X.25 Calls: API Level

The packets exchanged during an X.25 call are discussed in X.25 Calls Overview: Packet Level. This API-level overview tells you how to use the application programming interface for both switched and permanent virtual circuits. The following sections refer to the example programs to show you how the subroutines are used:

- X.25 API: Initializing and Terminating
- X.25 API: Using the Connection Identifier for Calls
- X.25 API: Using Counters to Correlate Messages
- X.25 API: Listening for Incoming Calls
- X.25 API: Making and Receiving a Call
- X.25 API: Transferring and Acknowledging Data
- X.25 API: Clearing, Resetting, and Interrupting Calls.

X.25 API: Initializing and Terminating

Initializing

The application programming interface (API) must be initialized for a specific X.25 port before any other subroutines can be used on that port. If the program uses more than one X.25 port, the API must be initialized for each. Use the **x25_init** subroutine (as in example program svcxmit).

Allocating a PVC

If the application is to use a permanent virtual circuit (PVC), you must use the **x25_pvc_alloc** subroutine to allocate the PVC, identifying it by its logical channel number and X.25 port name (as in example program pvcxmit). You can find out which logical channel numbers are valid by using the **smit** command.

Freeing a PVC

A permanent virtual circuit (PVC) must be freed using the **x25_pvc_free** subroutine, before the program is terminated (as in example program pvcxmit).

Terminating

You must terminate the API for each X.25 port, using the **x25_term** subroutine (as in example program svcxmit).

Before you terminate, however, there may be some tidying up to do:

- Clear any calls, using **x25_call_clear**
- Remove any counters, using **x25_ctr_remove**
- Stop listening for calls, using **x25_deafen**
- Free any permanent virtual circuits, using **x25_pvc_free**.

/dev/x25sn Special File

The **/dev/x25s***n* special file provides access to X.25 network adapters by way of the X.25 device handler. The handler provides multiple X.25 connections on each multiplexed channel.

The X.25 device handler may be loaded and unloaded. The handler supports configuration calls to initialize and terminate itself. Calls other than the **open** and **close** system calls are discussed based on the mode in which the device handler is operating.

The device handler supports the **/dev/x25s***n* special file as a character-multiplex special file. The special file must be opened for both reading and writing (O_RDWR). There are no particular considerations for closing the special file. The special file name used in an **open** call differs depending on how the device is to be opened. Types of special file names are:

/dev/x25s*n* Starts the device handler on the next available port.

/dev/x25s*n***/D** Starts the device handler in Diagnostic mode.

/dev/x25s*n***/M** Starts the device handler for reading and writing data to the monitor facilities on the IBM X.25 Interface Co-Processor/2.

/dev/x25s*n***/R** Starts the device handler for updating the routing table.

X.25 API: Using the Connection Identifier for Calls

Connection Identifier

Because the API, or even a single application, can be controlling more than one virtual circuit at a time, there must be a way of identifying a call uniquely. The API assigns to each call a positive integer known as the connection identifier.

The **conn_id** parameter is used by the API subroutines to pass the connection identifier.

Obtaining a Connection Identifier

On a switched virtual circuit, for an outgoing call, the connection identifier is returned by **x25_call** (as in example program svcxmit). When receiving an incoming call the connection identifier is allocated by **x25_receive** to the first of its parameters (as in example program svcrcv).

On a permanent virtual circuit, the connection identifier is returned by **x25_pvc_alloc** (as in example program pvcxmit).

Using a Connection Identifier

The connection identifier is assigned on return from the following subroutines:

- To make a call on a switched virtual circuit, using **x25_call**.
- To establish a permanent virtual circuit, using **x25_pvc_alloc**.
- To receive an incoming call, using **x25_receive**.
- To receive data from any currently connected call, using **x25_receive**.

The connection identifier is passed as a parameter to the following subroutines:

- To receive data from a particular call, using **x25_receive**.
- To accept a call, using **x25_call_accept**.
- To send data, using **x25_send**.
- To acknowledge data, using **x25_ack**.
- To reject or terminate a call, using **x25_call_clear**.
- To reset a call, using **x25_reset**.
- To confirm that a reset arrived, using **x25_reset_confirm**.
- To interrupt a call, using **x25_interrupt**.
- To free a permanent virtual circuit, using **x25_pvc_free**.
- To get information about a virtual circuit, using **x25_circuit_query**.

Restrictions on the Use of the Connection Identifier

A connection identifier can be used only by the process that made the call, established the permanent virtual circuit, or received the call, and its child processes. Any attempt to use the connection identifier of another process results in the **X25BADCONNID** error code.

X.25 API: Using Counters to Correlate Messages

Counters

Many applications can be using the network at once and each application may have several calls active at one time. An application may also be listening for calls for several different routing list names. How does the application know when a message has arrived on a particular virtual circuit, or for a particular call? A *counter*, supplied by the API, is incremented whenever a message arrives. The application issues an **x25_ctr_wait**, which returns when the counter has been incremented. The counter is decremented when the message has been received (using **x25_receive**).

Counters allow an application to wait for messages on several virtual circuits at one time; it is the responsibility of the application to correlate counters with particular virtual circuits. Optionally, an application can wait, using **x25_ctr_wait**, for several messages to accumulate against a particular counter before being notified.

Counter Identifiers

Each counter has a *counter identifier*; the **ctr_id** parameter is used by some of the API subroutines to pass the counter identifier. **x25_ctr_wait** uses an array of structures (**ctr_array_struct**) each of which contains a counter identifier and a value; this allows an application to wait for any of a number of counters to change.

How to Use Counters in Applications

It is up to you to decide how to use counters in your application, depending on what the application has to do. Use of counters is not required, but the use of **x25_ctr_wait** is the recommended way of notifying the application that a message has arrived.

For an application that makes calls, we recommend using a separate counter for each call. For an application that listens for and receives calls, use one counter to listen for incoming calls and then use a separate counter to accept each call and receive its subsequent messages. For an application that receives messages from any one of a number of connected calls, use a single counter.

The application is responsible for making sure that it gets enough counters.

Obtaining a Counter

The application gets a counter from the API, by calling the **x25_ctr_get** subroutine (as in example program svcxmit). This subroutine returns a counter identifier that is unique across the system.

The two applications-the one that makes a call and the one that receives it–each use a different counter for the call; they keep track of the messages independently of each other.

Using a Counter

The counter identifier is passed as a parameter to the following subroutines:

- **x25_call** assigns a counter to a specific connection, when making a call on a switched virtual circuit.
- **x25_pvc_alloc** assigns a counter to a specific connection, when establishing a permanent virtual circuit.
- **x25_listen** assigns a counter to a listening process, when starting to listen for incoming calls.
- **x25_call_accept** assigns a counter to a specific connection, when accepting a call.
- **x25_ctr_wait** uses an array of counters to wait for an incoming call or a message.
- **x25_ctr_test** use one counter to find out how many messages are waiting to be received for a call.

Waiting for an Incoming Call or a Message

Using **x25_ctr_wait** is the normal way in which an application program finds out that a message has arrived. You invoke **x25_ctr_wait** passing it a pointer to an array of counter structures. This enables an application to wait for messages for more than one call.

The example programs show **x25_ctr_wait** being used in several situations, but always with only one counter. If you want to wait for messages using multiple counters, you must assign them all to **ctr_array_struct** before invoking **x25_ctr_wait**.

Be aware that, if you are writing a program that uses multiple counters to identify multiple calls, you are responsible for storing the counter identifiers with their corresponding connection identifiers.

Examples of x25_ctr_wait

1. Set up the **ctr_array** and wait for an incoming call (as in example program svcrcv).
2. Wait for an acknowledgement, when the **ctr_array** was set up earlier in the program (as in example program svcxmit).

Finding Out How Many Messages are Waiting to be Received For a Call

The **x25_ctr_test** subroutine is provided for this purpose. The example shows how to use it:

1. Assign the counter identifier to **ctr_id**.
2. Invoke **x25_ctr_test**, passing **ctr_id** as a parameter.
3. The return value is the number of messages waiting to be received.

However, if you use **x25_ctr_wait** when you expect a message to arrive and receive every message when it arrives, you should not need to use **x25_ctr_test**.

Removing a Counter

Before an application terminates, it should remove all counters that were in use. A counter cannot be removed while its value is greater than 0, indicating that there is a message to be received. First, receive any messages and then use the **x25_ctr_remove** subroutine, passing it the counter identifier as a parameter (as in example program svcxmit).

Restrictions on the Use of Counters

Any application can test the value of a counter or wait for it to change. Only the application that requested the counter with **x25_ctr_get**, or a root user, can use the corresponding **x25_ctr_remove**.

X.25 API: Listening for Incoming Calls

Listen Identifier

The listen identifier is a positive integer returned by **x25_listen**. It is used by **x25_receive** to identify an incoming call. After the call has been received and accepted, the listen identifier continues to be used to listen for subsequent incoming calls.

Obtaining a Listen Identifier (Starting to Listen for Incoming Calls)

1. Get a counter.
2. Invoke the **x25_listen** subroutine, passing it two parameters: the counter identifier and the name of an entry in the X.25 routing list (as in example program svcrcv).
3. **x25_listen** returns a listen identifier.

Using the Listen Identifier

After obtaining a listen identifier, the application must wait for an incoming call. When an incoming call arrives for that listen identifier, the application assigns the listen identifier to the **conn_id** parameter and uses **x25_receive**, to receive the incoming call.

Removing the Listen Identifier (Stopping Listening)

1. Remove the counter associated with this listening process.
2. Invoke the **x25_deafen** subroutine, passing it the listen identifier as a parameter. Always do this before terminating a program that has been listening for incoming calls (as in example program svcrcv).

Restrictions on the Use of the Listen Identifier

The use of this variable is restricted to the *user* who received the **listen_id** from the **x25_listen** subroutine. The user may have one application that does the listening and notifies them of an incoming call, and another application that actually receives the call.

X.25 API: Making and Receiving a Call

Making an Outgoing Call

To make a call on a switched virtual circuit (SVC) (as in example program svcxmit):

1. Set up the **cb_call_struct** with the relevant information.
2. Invoke the **x25_call** subroutine, passing two parameters: a pointer to **cb_call_struct** and a counter identifier.
3. The **x25_call** subroutine returns a connection identifier, which the application must use to identify the call.
4. Store a counter identifier with the connection identifier.

If using a permanent virtual circuit (PVC), do not make any calls; after you have allocated a PVC, you can send and receive data until you free the PVC.

After Making a Call

The calling application must wait for the called application's response, using **x25_ctr_wait,** and then receive it, using **x25_receive** (as in example program svcxmit). The response could be either a call-connected message or the clear-indication message.

Receiving an Incoming Call

When you know that there is an incoming call waiting because the counter associated with the listen identifier has been incremented, you must use the listen identifier to receive the incoming call (as in example program svcrcv).

1. Assign the *listen identifier* to **conn_id.**
2. Invoke the **x25_receive** subroutine, passing two parameters:
   - The address of **conn_id** (which currently contains the listen identifier).
   - A pointer to the message control block, **cb_msg_struct**.
3. On return, **x25_receive** assigns the *connection identifier* of the incoming call to **conn_id**. (The listen identifier is still valid for further incoming calls.)
4. On return from **x25_receive**, the message control block includes the **msg_type**, which indicates the type of message. In this case it is `X25_INCOMING_CALL` and you do not need to check it because it is the only type of message that can be received using the listen identifier. The **cb_call_struct** control block contains the incoming-call message, which may include call user data.
5. Free any structures allocated by **x25_receive** (as in example program svcrcv).

Accepting or Rejecting an Incoming Call

After receiving an incoming call, you can accept it (as in example program svcrcv):

1. Get a new counter, to be used for accepting the call and receiving any subsequent messages for it. (This allows the counter that was used for listening to continue to be used to listen for calls.)
2. Optionally, set up **cb_call_struct** with the relevant information.
3. Invoke **x25_call_accept**, passing the connection identifier, the counter identifier, and **cb_call_struct** as parameters.
4. **x25_call_accept** sends an `X25_CALL_CONNECTED` message, which must be received by the caller.

At this point, after accepting a call, you should deal with the call user data if necessary. After dealing with the call user data, free the storage used (as in example program svcrcv).

Instead of receiving an incoming call, you can reject it, using **x25_call_clear** to clear it.

X.25 API: Transferring and Acknowledging Data

Sending Data

Either the called or the calling application can send data when:

- On a permanent virtual circuit (PVC), the PVC has been allocated.
- On a switched virtual circuit (SVC), a call has been made, received, and accepted.

To send data (as in example program svcxmit):

1. Ensure that any data sent previously with the D-bit set to 1 has been acknowledged. Otherwise this **x25_send** will fail.
2. Assign to the **data** variable in **cb_data_struct** a pointer to the data you want to send.
3. Assign to the **data_len** variable in **cb_data_struct** the length of the data.
4. Invoke the **x25_send** subroutine, passing two parameters: the connection identifier and a pointer to **cb_data_struct**.

Asking for Data to be Acknowledged by the Receiver

To ask for the receiver to acknowledge the data, set the flags to **X25FLG_DBIT** in **cb_data_struct**, before using **x25_send** (as in example program svcxmit). The application must then wait for and receive the `X25_DATA_ACK` message that is sent back. Note that to allow the use of the D-bit, it should also be set on **x25_call** (as in example program svcxmit) or **x25_call_accept**.

Long Messages

If the length of the data is greater than the packet size, the API automatically splits the data into packets which it sends separately. It sets on the M-bit in each packet to indicate that there is more data. Only the final packet has the D-bit set and only one acknowledgment is expected.

Nevertheless, you should try to avoid sending data longer than the packet size, to allow better recovery in the event of a transmission failure. Specify as large a packet size as possible in the maximum transmit packet size attribute (for an SVC) or PVC maximum transmit packet size. Or specify as large a packet size as possible in **psiz_cld** or **psiz_clg** in **cb_fac_struct**. If necessary, split up the data yourself in the application, if you want to receive an acknowledgment for each packet, and thus maintain data integrity. Otherwise, if one piece of the data does not arrive, all of the data may need to be sent again.

Receiving Data

To receive data that has arrived for *a particular call* (as in example program svcrcv):

1. Ensure that you have acknowledged any data received previously with the D-bit set to 1. Otherwise this **x25_receive** will return **X25NODATA.**
2. Invoke the **x25_receive** subroutine, passing the address of the connection identifier and the address of the message structure (**cb_msg_struct**) as parameters.
3. **x25_receive** receives a complete packet sequence. That is to say, if a long message was split up when it was sent, the X.25 API attempts to rebuild it before notifying the application that there is a message waiting. If any packet (other than data or interrupt) arrives before the sequence is completed, the attempt to rebuild is *either* abandoned and the sequence made available to the application up to its current position, *or* the incoming packet is made available to the application ahead of the as-yet-unfinished sequence.
4. On return from **x25_receive**, the message structure (**cb_msg_struct**) includes the **msg_type**, which indicates the type of message. In this case it is `X25_DATA`, indicating that the message is available in the **cb_data_struct** control block.
5. The counter that indicated the waiting message is decremented when the message is received.

To receive data from *any call that is currently connected*, assign 0 to **conn_id** and invoke the **x25_receive** subroutine, passing the address of **conn_id** as a parameter. On return from **x25_receive** the **conn_id** contains the connection identifier of the call whose data was returned by **x25_receive**.

Acknowledging Data Packets

For each data packet that was sent with the D-bit set to 1, invoke the **x25_ack** subroutine to confirm that it arrived (as in example program svcrcv).

The application should ensure that the acknowledgement is given as soon as possible after receiving a message with the D-bit set to 1.

X.25 API: Clearing, Resetting, and Interrupting Calls

Clearing a Call

Clearing does just that: it clears the call from the network. You can send a clear-request message for several different reasons.

Rejecting a Call

If you clear a call without ever accepting it, you are, in effect, rejecting it.

Receiving Fast-Select Data

If it is a fast-select call, the fast-select data is in the incoming-call packet. You can clear the call immediately after receiving this or you can receive further messages on the call.

Terminating a Call

Clearing is the normal way of terminating a call. Either the caller or the called application can clear a call.

To clear a call

1. Optionally, assign any data you want to send to the **user_data** field in the **cb_clear_struct** control block, and set the user-data flag
2. Optionally, assign a cause code and a diagnostic code to the appropriate fields in the **cb_clear_struct** control block, and set the appropriate flags.
3. Invoke **x25_call_clear**, passing the connection identifier and a pointer to **cb_clear_struct** as parameters. There is a third parameter that can be used for return data. If you do not need this, set the third parameter to **NULL.**

Example program svcxmit shows how a call is cleared. Note that example program svcrcv could have cleared the call after receiving the data; svcxmit is therefore prepared for the call to be cleared by the other application. A call does not have to be cleared by the application that made it.

Resetting a Call

A reset flushes any data being sent at the time from the network. To reset a call (as in example program pvcxmit):

1. Optionally, assign a cause code and a diagnostic code to the appropriate fields in the **cb_res_struct** control block and set the appropriate flags.
2. Invoke the **x25_reset** subroutine, passing it the connection identifier and a pointer to the **cb_res_struct** control block.
3. Wait for and receive the reset-confirmation message.

Handling a Reset

When an application receives a message with a **msg_type** of `X25_RESET_INDICATION`, it must send a reset-confirmation message immediately by invoking the **x25_reset_confirm** subroutine (as in example program pvcrcv).

Interrupting a Call

An interrupt is placed at the beginning of the queue of incoming messages. To send an interrupt:

1. Assign the connection identifier to **conn_id**.
2. Invoke the **x25_interrupt** subroutine, passing it the **conn_id** and a pointer to the **cb_int_data_struct** control block.
3. Wait for and receive the interrupt-confirmation message. (Using this X.25 API, the interrupt-confirmation is sent automatically.)

Related Information

The **open** system call, **close** system call.

Special Files Overview in *Files Reference*.

List of X.25 API Error Codes

For X.25-specific error conditions, **x25_errno** is set to one of the following values:

X25ACKREQ One or more packets require acknowledgement. Issue **x25_ack** before continuing.

X25AUTH The calling application does not have system permission to control the status of the link.

X25AUTHCTR The application does not have permission to remove this counter because it is not the application that issued the corresponding **x25_ctr_get**.

X25AUTHLISTEN The application cannot listen to this name, because the corresponding entry in the routing list has a user name that excludes the user running the application. Use another routing list name, or change the user name in the routing list entry.

X25BADCONNID The connection identifier is invalid.

X25BADDEVICE The X.25 port name is invalid.

X25BADID The connection identifier or listen identifier is invalid.

X25BADLISTENID The listen identifier is invalid.

X25CALLED The called address is invalid. Check that the address is correct and is a NULL-terminated string.

X25CALLING The calling address is invalid. Check that the address is correct and is a NULL-terminated string.

X25CTRUSE The counter has a non-zero value.

X25INIT X.25 is already initialized for this X.25 port, so cannot be initialized again.

X25INVCTR The specified counter does not exist. (In the case of **x25_ctr_wait**, the counter is one of an array of counters.)

X25INVFAC An optional facility requested is invalid. Check **cb_fac_struct**.

X25INVMON The monitoring mode is invalid.

X25LINKUP The X.25 port is already connected.

X25LINKUSE The X.25 port still has virtual circuits established; it may still be in use. Either free all virtual circuits or disconnect the port using the override.

X25LONG The parameter is too long. Check each of the parameters for this subroutine.

X25MAXDEVICE Attempts have been made to connect more X.25 ports than are available. Check the **smit** configuration to see how many ports are available.

X25MONITOR X.25 traffic on this X.25 port is already being monitored by another application. The other application must stop monitoring before any other application can start it.

X25NAMEUSED Calls for this name are already being listened for.

X25NOACKREQ No packets currently require acknowledgement.

X25NOCARD The X.25 adapter is either not installed or not functioning.

X25NOCTRS No counters are available.

X25NODATA No data is has arrived for this connection identifier. Issue **x25_ctr_wait** to be notified when data arrives.

X25NODEVICE The X.25 device driver is either not installed or not functioning.

X25NOLINK The X.25 port is not connected. Issue **x25_link_connect**, or use **xmanage** to connect it.

X25NONAME The name is not in the routing list. Add the name or use one that is already in the list.

X25NOSUCHLINK The X.25 port does not exist. Check the **smit** configuration.

X25NOTINIT The application has not initialized X.25 communications. Issue **x25_init**.

X25NOTPVC This is not defined as a permanent virtual circuit (PVC). Check the **smit** configuration.

X25PROTOCOL An X.25 protocol error occurred.

X25PVCUSED This permanent virtual circuit (PVC) is already allocated to another application. The other application must free the PVC before it can be used.

X25RESETCLEAR The call was reset or cleared during processing. Issue **x25_receive** to obtain the reset-indication or clear-indication packet. Then issue **x25_reset_confirm** or **x25_clear_confirm**, as necessary.

X25SYSERR An error occurred that was not an X.25 error. Check the value of **errno**.

X25TABLE The routing list cannot be updated because **xroute** is using it. Try again after **xroute** has completed.

X25TIMEOUT A timeout problem occurred.

X25TOOMANYVCS No virtual circuits are free on the listed X.25 ports.

X25TRUNCTX The packet size is too big for internal buffers, so data cannot be sent.

List of System Error Codes

For non-X.25-specific error conditions, **x25_errno** is set to X25SYSERR and **errno** is set to one of the following values:

EFAULT Bad address pointer.

EINTR A signal was caught during the call.

EIO An I/O error occurred.

ENOMEM Could not allocate memory for device information.

ENOSPC There are no buffers available in the pool.

EPERM Calling application does not have sufficient authorization.

X.25 Example Programs Overview

To help you learn how to use the subroutines there are two pairs of example programs; one pair demonstrates the use of a switched virtual circuit, and the other the use of a permanent virtual circuit. The SVC call is similar to the example call described in X.25 Calls Overview: Packet Level.

- X.25 Example Program svcxmit: Make a Call Using an SVC
- X.25 Example Program svcrcv: Receive a Call Using an SVC
- X.25 Example Program pvcxmit: Send Data Using a PVC
- X.25 Example Program pvcrcv: Receive Data Using a PVC.

Preparing, Compiling, and Running the Example Programs

The example programs (svcxmit.c, svcrcv.c, pvcxmit.c, and pvcrcv.c) are in **/usr/lpp/bosext2/x25app/samples**.

Each of the example programs has some variables to which values are assigned at the start; these include **CALLING_ADDR, CALLED_ADDR, LINK_NAME**, and **LOG_CHAN_NUM,** which should be set to appropriate values for your setup, before you can run the programs.

To compile the example programs, type:

```
cd /usr/lpp/bosext2/x25app/samples
cc svcxmit.c -lx25s
cc svcrcv.c -lx25s
cc pvcxmit.c -lx25s
cc pvcrcv.c -lx25s
```

This creates the executable files, **svcxmit**, **svcrcv**, **pvcxmit**, and **pvcrcv**.

To run a program, type the name of the executable file at the shell prompt. Run them in pairs: svcxmit talks to svcrcv and pvcxmit talks to pvcrcv. Note that you cannot run the PVC programs unless your network allows the use of permanent virtual circuits.

Using the Example Code

Note that the example programs are for demonstration purposes only. When creating your own programs, you may find it useful to copy parts of the code from the examples. Be aware that the examples do not, in most cases, check the return codes from the subroutines. When you invoke an X.25 subroutine in a production program, you should assign the return value into a variable, like this:

```
rc = x25_...(...);
```

Then test the value of the return code.

If you do not want to write your own programs, use the **xtalk** command to communicate with other people. Use the other X.25 commands to manage the X.25 network.

X.25 Example Program pvcrcv: Program Description

This program uses a permanent virtual circuit (PVC).

1. Initialize the API for the port specified by **LINK_NAME** (**x25_init**).
2. If initialization failed, display a message and exit from the program.
3. Get a counter to be used to wait for incoming messages (**x25_ctr_get**).
4. Allocate a PVC to the port, using the logical channel number specified by **LOG_CHAN_NUM** (**x25_pvc_alloc**).
5. If PVC allocation failed, display a message and exit from the program.

Do until the end–of–transmission indicator is received:

1. Wait for an incoming message (**x25_ctr_wait**).
2. Receive the incoming message (**x25_receive**).
3. Test the **msg_type** in **cb_msg_struct**:
   a. If the incoming message is a reset indication, send a reset confirmation (**x25_reset_confirm**).
   b. If the incoming message is data display it on the screen. (If it is the end-of-tranmission indicator specified in **END_OF_TRANS**, print a message saying that transmission has ended.) Free the storage allocated to **cb_msg_struct**.

When the end–of–transmission indicator has been received:

1. Free the permanent virtual circuit (**x25_pvc_free**).
   a. Remove the counter (**x25_ctr_remove**).
   b. Terminate the API for port x25s1 (**x25_term**).

X.25 Example Program pvcrcv: Receive Data Using a PVC

```
/* X.25 Example Program pvcrcv. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <NLchar.h>
#include <x25sdefs.h>

#define LINK_NAME     "x25s0"      /* Name of X.25 port.              */
#define LOG_CHAN_NUM  (1)         /* PVC logical channel number.     */
#define END_OF_TRANS "EOP"        /* End-of-transmission indicator; */
                                  /* must be the same as in pvcxmit.

/*******************************************************************/
/* Function     main                                               */
/* Description  This program is designed to demonstrate usage     */
/*   of the X.25  API.                                             */
/*  It allocates a permanent virtual circuit,                      */
/*   receives data and                                             */
/*  is prepared to handle a reset, by sending a                    */
/*   reset-confirmation.                                           */
/*  Example Program pvcxmit is designed to send the data received */
/*   By this program.                                              */
/*   Note that, in a production program, you should check the      */
/*   return code from each subroutine call and take appropriate    */
/*   action.                                                       */
/*   Return       0 if successful                                  */
/*                1 otherwise                                      */
/*******************************************************************/

int main(
  int argc,
  char *argv[])
{

/*******************************************************************/
/* The following structures are defined in the x25sdefs.h file. */

/*******************************************************************/
  struct ctr_array_struct ctr_array[1];
                 /* One counter in the array.                    */
  struct cb_msg_struct cb_msg;

  struct cb_pvc_alloc_struct cb_pvc;

  struct cb_link_name_struct cb_link_name;
```

```
  int conn_id;
        /* Connection identifier to associate with this link.*/
  int ctr_id;
        /* Counter identifier for this link.                 */
  int rc;
        /* Return codes from various subroutines.            */
  int ctr_num = 1;
        /* Number of counters in the counter array.          */
  int end_tx = 0;
        /* Whether end of transmission has been reached.     */
/****************************************************************/
/*  Initialize the API for access to a link.                 */
/****************************************************************/
  cb_link_name.flags = X25FLG_LINK_NAME;
  cb_link_name.link_name = LINK_NAME;
  rc = x25_init(&cb_link_name);
  if (rc < 0)
  {
   (void)printf("%s: x25_init failed : x25_errno = %d errno = %d\n",
      argv[0],x25_errno,errno);
    return(1);
  }
  else
  {
/*****************************************************************/
 /* Get a counter to be used to notify us of incoming messages.*/

/*****************************************************************/
    ctr_id = x25_ctr_get();

/*****************************************************************/
/* Set up flags to show that a link and a channel number are   */
/* supplied.                                                   */
/* Then allocate the permanent virtual circuit for this        */
/* application.                                                */

/****************************************************************/
    cb_pvc.flags = X25FLG_LINK_NAME | X25FLG_LCN;
    cb_pvc.link_name = LINK_NAME;
    cb_pvc.lcn = LOG_CHAN_NUM;

    conn_id = x25_pvc_alloc(&cb_pvc,ctr_id);

    if (conn_id < 0)
    {
      (void)printf("%s: x25_pvc_alloc failed : x25_errno = %d errno
                 = %d\n", argv[0],x25_errno,errno);
       return(1);
    }
    else
    {
```

```
/*****************************************************************/
 /* The PVC link has now been set up and data can be received.  */
 /* Wait for any message to arrive for this application         */

/****************************************************************/
      ctr_array[0].flags = X25FLG_CTR_ID;
      ctr_array[0].flags |= X25FLG_CTR_VALUE;
      ctr_array[0].ctr_id = ctr_id;
      ctr_array[0].ctr_value = 0;

      do
      {
        (void)x25_ctr_wait(ctr_num,ctr_array);
/*****************************************************************/
        /* Receive the message                                   */

/****************************************************************/
        (void)x25_receive(&conn_id,&cb_msg);
/*****************************************************************/
      /* If a reset-indication message is received, we must     */
      /* send a reset-confirmation message as soon as possible. */

/*****************************************************************/
        if (cb_msg.msg_type == X25_RESET_INDICATION)
        {
        (void)printf("%s: Received reset indication...",argv[0]);
          (void)x25_reset_confirm(conn_id);
        }
 /*If data is received, we display it on the screen, unless it is */
  /* end-of-transmission indicator specified by END_OF_TRANS.      */

     else if (cb_msg.msg_type == X25_DATA)
     {
       (void)printf("%s: Incoming Data : ",argv[0]);
       (void)printf("%s\n",cb_msg.msg_point.cb_data->data);
       if (strcmp(cb_msg.msg_point.cb_data->data,END_OF_TRANS) != 0)
          {
            (void)printf("%s",cb_msg.msg_point.cb_data->data);
            (void)printf("\n");
          }
          else
          {
           (void)printf("%s: End of transmission received",argv[0]);
            end_tx = 1;
          }
```

```
/*****************************************************************/
/* The X.25 API allocates memory for information to be returned. */
/* Although there are no memory constraints in this application, */
/* the space is freed when the information has been displayed.   */

/*****************************************************************/
         free((char *)cb_msg.msg_point.cb_data->data);
         free((char *)cb_msg.msg_point.cb_data);
       }
       else
       {
         (void)printf("%s: Unexpected packet received",argv[0]);
       }
      }  while (end_tx == 0);

/*****************************************************************/
/*Free up any resources allocated during the program before    */
/*   ending: free the permanent virtual circuit                */
/*   remove the counter                                        */
/*   terminate the API.                                        */

/*****************************************************************/
      (void)x25_pvc_free(conn_id);
      (void)x25_ctr_remove(ctr_id);
      (void)x25_term(&cb_link_name);
     }
  }
  return(0);
}
```

X.25 Example Program pvcxmit: Program Description

This program uses a permanent virtual circuit (PVC).

1. Initialize the API for the port specified by **LINK_NAME** (**x25_init**).
2. If initialization failed, display a message and exit from the program.
3. Get a counter to be used to wait for incoming messages (**x25_ctr_get**).
4. Allocate a PVC to the port, using the logical channel number specified by **LOG_CHAN_NUM** (**x25_pvc_alloc**).
5. If PVC allocation failed, display a message and exit from the program.
6. Send some data (**x25_send**).
7. Send a reset (**x25_reset**).
8. Wait for the reset-confirmation message (**x25_ctr_wait**).
9. Receive the reset-confirmation message (**x25_receive**).
10. Send some more data (**x25_send**).
11. Send the end-of-transmission indicator specified by **END_OF_TRANS** (**x25_send**).
12. Free the permanent virtual circuit (**x25_pvc_free**).
13. Remove the counter (**x25_ctr_remove**).
14. Terminate the API for the port (**x25_term**).

X.25 Example Program pvcxmit: Send Data Using a PVC

Example Program pvcxmit

```
/* X.25 Example Program pvcxmit. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <NLchar.h>
#include <x25sdefs.h>

#define LINK_NAME    "x25s0"
                     /* Name of X.25 port.                */
#define LOG_CHAN_NUM  (1)
                     /* PVC logical channel number.       */

#define INFO  "Hello World"
                      /* Data to be sent.                 */
#define INFO2 "Goodbye Everyone"
                      /* More data to be sent.            */
#define END_OF_TRANS "EOP"
                      /* End-of-transmission indicator:   */

                       /* must be the same as in pvcrcv.   */

/*****************************************************************/
/* Function     main                                             */
/* Description  This program is designed to demonstrate          */
/* usage of the X.25 API.                                        */
/* It allocates a permanent virtual circuit, sends some data     */
/* and then sends a reset.  After receiving the                  */
/* reset-confirmation, the program sends some more data.         */
/* Example Program pvcrcv is designed to receive the data sent   */
/* by this program.                                              */
/* Note that, in a production program, you should check the      */
/* return code from each subroutine call and take appropriate    */
/* action.                                                       */
/* Return       0 if successful                                  */
/*              1 otherwise                                      */
/*****************************************************************/
int main(
  int argc,
  char *argv[])
{
/*****************************************************************/
/* The following structures are defined in the x25sdefs.h file. */

/*****************************************************************/
  struct ctr_array_struct ctr_array[1];
                    /* One counter in the array. */
  struct cb_msg_struct cb_msg;
  struct cb_pvc_alloc_struct cb_pvc;
  struct cb_res_struct cb_res;
  struct cb_link_name_struct cb_link_name;
  struct cb_data_struct cb_data;
```

```
  int conn_id;
           /* Connection identifier to associate with this link.*/
  int ctr_id;
            /* Counter identifier for this link.             */
  int ctr_num = 1;
           /* Number of counters in the counter array.       */
  int rc;
           /* Return codes from various subroutines.         */
/*******************************************************************/
  /* Initialize the API for access to a link.                     */
/*******************************************************************/
  cb_link_name.flags = X25FLG_LINK_NAME;
  cb_link_name.link_name = LINK_NAME;
  rc = x25_init(&cb_link_name);
  if (rc < 0)
  {
   (void)printf("%s: x25_init failed : x25_errno = %d errno = %d\n",
      argv[0],x25_errno,errno);
    return(1);
  }
  else
  {
/*******************************************************************/
/* Get a counter to be used to notify us of incoming messages.    */

/******************************************************************/
    ctr_id = x25_ctr_get();

/*******************************************************************/
/* Set up flags to show that a link and a channel number are      */
/* supplied.                                                      */
/* Then allocate the permanent virtual circuit for this           */
/* application.                                                   */

/*******************************************************************/
    cb_pvc.flags = X25FLG_LINK_NAME | X25FLG_LCN;
    cb_pvc.link_name = LINK_NAME;
    cb_pvc.lcn = LOG_CHAN_NUM;

    conn_id = x25_pvc_alloc(&cb_pvc,ctr_id);

    if (conn_id < 0)
    {
      (void)printf("%s: x25_pvc_alloc failed : x25_errno = %d errno
                  = %d\n", argv[0],x25_errno,errno);
       return(1);
    }
    else
    {
```

```
/*****************************************************************/
/* Now the PVC is available, send some data.                    */

/****************************************************************/
      cb_data.flags = X25FLG_DATA;
      cb_data.data_len = strlen(INFO);
      cb_data.data = INFO;
      (void)printf("%s: Sending some data...",argv[0]);
      (void)x25_send(conn_id,&cb_data);

/*****************************************************************/
 /* Send a reset.                                               */

/****************************************************************/
      (void)printf("%s: Resetting the circuit...",argv[0]);
      (void)x25_reset(conn_id,&cb_res);

/*****************************************************************/
/* After sending a reset packet, you must wait for the reset    */
/* confirm to arrive.                                            */

/****************************************************************/
      ctr_array[0].flags = X25FLG_CTR_ID;
      ctr_array[0].flags |= X25FLG_CTR_VALUE;
      ctr_array[0].ctr_id = ctr_id;
      (void)x25_ctr_wait(ctr_num,ctr_array);

/*****************************************************************/
 /* There is now a message ready to be received.  If it is      */
 /* anything other than the expected reset-confirmation, we:    */
 /* free the permanent virtual circuit  remove the counter.     */
 /* terminate the API.                                           */

/*****************************************************************/
      (void)x25_receive(&conn_id,&cb_msg);

      if (cb_msg.msg_type != X25_RESET_CONFIRM)
      {
        (void)printf("%s: Did not receive expected reset
confirm",argv[0]);
        (void)x25_pvc_free(conn_id);
        (void)x25_ctr_remove(ctr_id);
        (void)x25_term(&cb_link_name);
        return(1);
      }

      (void)printf("%s: Received reset confirm...",argv[0]);

/*****************************************************************/
 /* Now send some more data                                       */
 /* The last block of data to be sent is the end-of-transmission  */
 /*indicator specified by END_OF_TRANS. This is understood by the */
 /* PVC receiver example program, pvcrcv.                        */

/*****************************************************************/
      cb_data.data_len = strlen(INFO2);
      cb_data.data = INFO2;
```

```
    (void)printf("%s: Sending some data...",argv[0]);
     (void)x25_send(conn_id,&cb_data);

     (void)printf("%s: Sending last block of data...",argv[0]);
     cb_data.data_len = strlen(END_OF_TRANS);
     cb_data.data = END_OF_TRANS;
     (void)x25_send(conn_id,&cb_data);
/*****************************************************************/
/* Free up any resources allocated during the program before    */
/*  ending: free the permanent virtual circuit                  */
/*   remove the counter.                                        */
/*   terminate the API.                                         */

/*****************************************************************/
     (void)x25_pvc_free(conn_id);
     (void)x25_ctr_remove(ctr_id);
     (void)x25_term(&cb_link_name);
   }
 }
 return(0);
```

X.25 Example Program svcrcv: Program Description

1. Initialize the API for the port specified by **LINK_NAME** (**x25_init**).
2. If initialization failed, display a message and exit from the program.
3. Get a counter for listening for incoming calls (**x25_ctr_get**).
4. Start listening for incoming calls (**x25_listen**).
5. Wait for an incoming call (**x25_ctr_wait**).
6. Receive the incoming call (**x25_receive**).
7. Get a counter for handling this call (**x25_ctr_get**).
8. Accept the call (**x25_call_accept**).
9. Free any memory allocated by the API to **cb_msg_struct**.

Repeat until a clear-indication message arrives:

1. Wait for a message (**x25_ctr_wait**).
2. Receive message (**x25_receive**).
3. If the message is data:
   a. Acknowledge it if the D-bit is set (**x25_ack**).
   b. Display the data on the screen.
   c. Free any memory allocated to **cb_msg_struct** by the API.
4. If the message is a clear-indication;
   a. Display a message to say the call has been cleared.
   b. Remove the counter (**x25_ctr_remove**).
   c. Stop listening for calls (**x25_deafen**).
   d. Terminate the API for the port (**x25_term**).
5. If the message is a reset-indication, send a reset-confirm (**x25_reset_confirm**).
6. If it is any other message type, do nothing.

X.25 Example Program svcrcv: Receive a Call Using an SVC

```
/* X.25 Example Program svcrcv. */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <NLchar.h>
#include <x25sdefs.h>

#define LINK_NAME    "x25s0"              /* Name of X.25 port.
        */
#define SAMPLE_NAME "IBMSAMP"             /* A name in the X.25
routing list.  */

/*****************************************************************/
/* Function     main                                             */
/* Description  This program is designed to demonstrate usage */
/* of the X.25 API. It waits for an incoming call, accepts it,*/
/* and then prints any data received.                           */
/* Example program svcxmit is designed to send the data        */
/* received by this program.                                    */
/* Note that, in a production program, you should check the    */
/* return code from each subroutine call and take appropriate */
/* action.                                                      */
/* Returns       0 if successful                                */
/*               1 if error                                     */
/******************************************************************/

int main(
  int argc,
  char *argv[])
{

/******************************************************************/
/* The following structures are defined in the x25sdefs.h file.*/

/******************************************************************/
  struct cb_call_struct cb_call;
  struct ctr_array_struct ctr_array[1];
                             /* This program waits for only */

                             /* one counter at a time.*/
  struct cb_msg_struct cb_msg;
  struct cb_link_name_struct cb_link_name;
```

```
  NLchar name[8];
         /* 1 longer than SAMPLE_NAME for NULL terminator.     */
  int listen_id;
         /* Listen identifier for x25_receive.                 */
  int conn_id;
         /* Connection identifier to identify the call after   */
          /* receiving it.                                     */
  int listen_ctr_id;
         /* Counter identifier to associate with incoming calls*/
  int call_ctr_id;
          /* Counter identifier to associate with accepted call */
  int ctr_num;
         /* Number of entries in ctr_array.                    */
  int rc;
          /* Return code                                        */
/*****************************************************************/
/* Initialize the API for access to a link.                     */

/*****************************************************************/
  cb_link_name.flags = X25FLG_LINK_NAME;
  cb_link_name.link_name = LINK_NAME;

  rc = x25_init(&cb_link_name);
  if (rc < 0)
  {
    (void)printf("%s: x25_init failed : x25_errno = %d errno =
               %d\n", argv[0],x25_errno,errno);
     return(1);
  }
  else
  {
/*****************************************************************/
 /* Prepare to receive incoming calls:                          */
 /* 1. Get a counter to be used to notify us of incoming calls. */
 /* 2. Listen for calls that satisfy the criteria specified by  */
 /* a name in the routing list.                                 */

/*****************************************************************/
    listen_ctr_id = x25_ctr_get();
                                 /* Get a counter.               */

    (void)NCdecstr(SAMPLE_NAME,name,8);
                                 /* Convert to NLchar.           */
    listen_id = x25_listen(name,listen_ctr_id);
    if (listen_id < 0)
    {
      (void)printf("%s: x25_listen failed : x25_errno = %d errno =
                 %d\n", argv[0],x25_errno,errno);
       return(1);
    }
    else
      (void)printf("%s: Awaiting incoming call...\n",argv[0]);
```

```
/*****************************************************************/
 /* Wait for an incoming call. The x25_ctr_wait subroutine      */
 /* returns when a message arrives.                             */

/****************************************************************/
    ctr_num = 1;
    ctr_array[0].flags = X25FLG_CTR_ID;
    ctr_array[0].flags |= X25FLG_CTR_VALUE;
    ctr_array[0].ctr_id = listen_ctr_id;
    ctr_array[0].ctr_value = 0;
    rc = x25_ctr_wait(ctr_num,ctr_array);

*****************************************************************/
 /* Receive an incoming call.                                   */
 /* In this example, we can assume that the message that has    */
 /* arrived (causing the counter to be incremented and          */
 /* x25_ctr_wait to return) is an incoming-call message.        */
 /* Therefore we assign the listen identifier to the conn_id    */
 /*parameter before invoking x25_receive and                    */
 /* we do not check the return code.                            */


 /* On return, conn_id is set to the connection identifier     */
  /* for this call.                                            */

/**************************************************************/

    conn_id = listen_id;
    (void)x25_receive(&conn_id,&cb_msg);

    (void)printf("%s: Incoming call received\n",argv[0]);


/*****************************************************************
/
/* Get a new counter for handling data from this call before     */
/* accepting the call.                                           */
/* No additional information needs to be put into the call-accept*/
/* packet, so the flags field is set to zero.                    */

/*****************************************************************/
    call_ctr_id = x25_ctr_get();
    cb_call.flags = 0;
    (void)x25_call_accept(conn_id,&cb_call,call_ctr_id);
    (void)printf("%s: Call accepted.\n",argv[0]);
```

```
/*****************************************************************/
/*  x25_receive allocates storage to return information. Although */
/*  there are no storage constraints in this application,          */
/*  the allocated storage is freed once the information           */
/*  is no longer needed.                                           */

/*****************************************************************/
    if (cb_msg.msg_point.cb_call != NULL)
    {
      cb_msg.msg_point.cb_call -> flags = 0;
      if (cb_msg.msg_point.cb_call->link_name != NULL)
        free(cb_msg.msg_point.cb_call->link_name);

      if (cb_msg.msg_point.cb_call->calling_addr != NULL)
        free(cb_msg.msg_point.cb_call->calling_addr);

      if (cb_msg.msg_point.cb_call->called_addr != NULL)
        free(cb_msg.msg_point.cb_call->called_addr);

      if (cb_msg.msg_point.cb_call->user_data != NULL)
        free(cb_msg.msg_point.cb_call->user_data);

      free(cb_msg.msg_point.cb_call);
    }

/*****************************************************************/
/* The call has now been received and accepted. Now wait for   */
/* the data.                                                    */

/*****************************************************************/
    do
    {

/*****************************************************************/
/* Wait for counter to indicate that data is waiting to be    */
/* received.                                                  */

/*****************************************************************/
      ctr_num = 1;
      ctr_array[0].flags = X25FLG_CTR_ID;
      ctr_array[0].flags |= X25FLG_CTR_VALUE;
      ctr_array[0].ctr_id = call_ctr_id;
      ctr_array[0].ctr_value = 0;
      (void)x25_ctr_wait(ctr_num,ctr_array);

/*****************************************************************/
/* Receive the message that is now ready. The types of message */
/* that the program can handle are data, clear-indication, and  */
/* reset-indication; other message types are ignored.          */

/*****************************************************************/
      (void)x25_receive(&conn_id,&cb_msg);

      switch (cb_msg.msg_type)
      {
      case X25_DATA:
```

```
/*****************************************************************/
/* Acknowledge the data if the D-bit (delivery confirmation)   */
/* is set.*/

/*****************************************************************/
        if ((cb_msg.msg_point.cb_data->flags) & X25FLG_D_BIT)
            (void)x25_ack(conn_id);

/*****************************************************************/
 /* Print the received data. Assume it is a normal string.     */

/*****************************************************************/
        if ((cb_msg.msg_point.cb_data -> flags) & X25FLG_DATA)
        {
          (void)printf("%s: Incoming Data : ",argv[0]);
          (void)printf("%s\n",cb_msg.msg_point.cb_data->data);
          free(cb_msg.msg_point.cb_data->data);
                                     /* Free memory allocated */
          free(cb_msg.msg_point.cb_data);
        }
        break;

      case X25_CLEAR_INDICATION:

/*****************************************************************/
/* When the call has been cleared, do the tidying up:          */
/*   Remove the counters.                                      */
/*   Stop listening for calls.                                 */
/*   Terminate the API.                                        */

/*****************************************************************/
        (void)printf("%s: Call cleared. Cause = 0x%02x Diagnostic =
                     0x%02x\n", argv[0],
                       cb_msg.msg_point.cb_clear->cause,
                       cb_msg.msg_point.cb_clear->diagnostic);
        (void)x25_ctr_remove(call_ctr_id);
        (void)x25_ctr_remove(listen_ctr_id);
        (void)x25_deafen(listen_id);
        (void)x25_term(&cb_link_name);
        break;

      case X25_RESET_INDICATION:

/*****************************************************************/
/* Respond to the arrival of a reset-indication message, by    */
/* sending a reset-confirmation message.                       */

/*****************************************************************/
        (void)x25_reset_confirm(conn_id);
        break;

     default:
 /* Ignore packet types other than data, clear-indication, and   */
 /* reset-indication.                                            */
         break;
      }
    } while (cb_msg.msg_type != X25_CLEAR_INDICATION);
  }
  return(0);
}
```

X.25 Example Program svcxmit: Program Description

This example program uses a switched virtual circuit:

1. Initialize the API for the port specified by **LINK_NAME** (**x25_init**).
2. If initialization failed, display a message and exit from the program.
3. Get a counter (**x25_ctr_get**).
4. Make a call from address specified by **CALLING_ADDR** to address specified by **CALLED_ADDR,** enabling D-bit acknowledgment. (**x25_call**)
5. Wait for a call-clear or call-connected message (**x25_ctr_wait**)
6. Receive the message (**x25_receive**)
7. If the message is call-connected:
    a. Send data (**x25_send**), without the D-bit set.
    b. Send data (**x25_send**), with the D-bit set.
    c. Wait for (**x25_ctr_wait**) and receive (**x25_receive**) acknowledgment of the data sent with the D-bit set.
    d. Clear the call (**x25_call_clear**).
8. If the call was cleared by the remote DTE (the other user), display a message.
9. Remove the counter (**x25_ctr_remove**).
10. Terminate the API (**x25_term**).

X.25 Example Program svcxmit: Make a Call Using an SVC

```
/* X.25 Example Program svcxmit. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <NLchar.h>
#include <x25sdefs.h>

#define LINK_NAME    "x25s0"
                              /* Name of X.25 port.          */
#define CALLING_ADDR "54321"
                              /* Calling Network User Address */
#define CALLED_ADDR  "1234502"
                              /* Called  Network User Address */
#define SAMPLE_NAME "IBMSAMP"
                          /* A name in the X.25 routing list. */

#define INFO        "Hello World"
#define INFO2       "Goodbye Everyone"

/*****************************************************************/
/* Function     main                                            */
/* Description  This program is designed to demonstrate usage  */
/* of the X.25 API. It makes a call, transmits some data,       */
/* and then clears the call.                                     */
/* Example program svcrcv is designed to receive the data sent  */
/* by this program.                                              */
/* Note that, in a production program, you should check the     */
/* return code from each subroutine call and take appropriate   */
/* action.                                                       */
/* Return       0 if successful                                  */
/*              1 if error occurs                                */
/*****************************************************************/
int main(
  int argc,
  char *argv[])
{
  int conn_id;
         /* Connection identifier,                              */
         /* to identify the call once it is made.               */
  int ctr_id;
        /* Counter identifier to be associated with the call. */
  int rc;
         /* Used for return codes.                              */
  int ctr_num = 1;
        /* The number of counters in counter array.            */
```

```
/*****************************************************************/
 /* The following structures are defined in the x25sdefs.h file. */
 /*  file.                                                       */

/*****************************************************************/
  struct cb_msg_struct cb_msg;

  struct cb_link_name_struct cb_link_name;

  struct ctr_array_struct ctr_array[1];

  struct cb_call_struct cb_call;

  struct cb_clear_struct cb_clear;

  struct cb_data_struct cb_data;
/*****************************************************************/
 /* Initialize the API for access to a link.                    */

/*****************************************************************/
  cb_link_name.flags = X25FLG_LINK_NAME;
  cb_link_name.link_name = LINK_NAME;
  rc = x25_init(&cb_link_name);

  if (rc < 0)
  {
    (void)printf("%s: x25_init failed : x25_errno = %d errno =
                %d\n",argv[0],x25_errno,errno);
    return(1);
  }
  else
  {

/*****************************************************************/
/* Get a counter to be used to notify us of incoming messages. */

/*****************************************************************/
     ctr_id = x25_ctr_get();

/*****************************************************************/
/* Set the flags in the cb_call structure to indicate which     */
/* fields have been filled in. The fields which this program    */
/* sets are the calling and called addresses, and the link on   */
/* which to call. The D-bit field must also be set, as there    */
/* will be a data packet sent later which sets the D-bit.       */

/*****************************************************************/
    cb_call.flags = X25FLG_LINK_NAME;
                              /* Set flag for using linkname.  */
    cb_call.link_name = LINK_NAME;

    cb_call.flags |= X25FLG_CALLING_ADDR;
                              /* Set flag for calling address. */
    cb_call.calling_addr = CALLING_ADDR;

    cb_call.flags |= X25FLG_CALLED_ADDR;
                              /* Set flag for called address.  */
    cb_call.called_addr = CALLED_ADDR;
```

```
    cb_call.flags |= X25FLG_D_BIT;
                               /* Set flag for D-bit.          */

/* Now that cb_call structure has been set up, make the call.   */
/* The return code is the connection identifier, which will be  */
 /* used to refer to this call later.                           */

    conn_id = x25_call(&cb_call,ctr_id);
    if (conn_id == -1)
    {
   (void)printf("%s: x25_call failed : x25_errno = %d errno = %d\n",
          argv[0],x25_errno,errno);
      return(1);
    }
    else
      (void)printf("%s: Placed outgoing call\n",argv[0]);

/*****************************************************************/
/* After making the call, prepare for either a call-connected    */
/* or aclear-indication message to arrive:                       */
/* wait for the counter value to change indicating an incoming   */
/* message. (If there were more than one counter in the array,   */
/* you would have to test the counter identifier to see which    */
/* one had been incremented. In this case there is only one,     */
/* so we do not have to do this.)                                */

/*****************************************************************/
    ctr_array[0].flags = X25FLG_CTR_ID;
    ctr_array[0].flags |= X25FLG_CTR_VALUE;
    ctr_array[0].ctr_id = ctr_id;
    ctr_array[0].ctr_value = 0;

    (void)x25_ctr_wait(ctr_num,ctr_array);

    /* Receive the call-clear or call-connected packet.
        */
    (void)x25_receive(&conn_id,&cb_msg);

/****************************************************************/
/* If the incoming message shows that the call has been        */
/* connected, send some data.                                  */

/*****************************************************************/
    if (cb_msg.msg_type == X25_CALL_CONNECTED)
    {
      cb_data.flags = X25FLG_DATA;
      cb_data.data_len = strlen(INFO);
      cb_data.data = INFO;
      (void)x25_send(conn_id,&cb_data);
      (void)printf("%s: Data sent\n",argv[0]);
```

```
/*****************************************************************/
/* Send some more data but this time with the D bit set. This    */
/* requires the receiver to send an acknowledgement to this      */
/* data, so we have to wait for the acknowledgment to arrive.    */

/*****************************************************************/
      cb_data.flags = X25FLG_DATA;
      cb_data.flags |= X25FLG_D_BIT;
      cb_data.data_len = strlen(INFO2);
      cb_data.data = INFO2;
      (void)x25_send(conn_id,&cb_data);
      (void)printf("%s: Data sent\n",argv[0]);

/* Wait for and receive acknowledgement                         */
      (void)x25_ctr_wait(ctr_num,ctr_array);
      (void)x25_receive(&conn_id,&cb_msg);

      if (cb_msg.msg_type == X25_DATA_ACK)
        (void)printf("%s: Data has been acknowledged.\n",argv[0]);
      else
        (void)printf("%s: Unexpected packet received.\n",argv[0]);


/*****************************************************************/
/* Clear the call now that transmission is completed.           */

/*****************************************************************/
      cb_clear.flags = X25FLG_CAUSE;
      cb_clear.flags |= X25FLG_DIAGNOSTIC;
      cb_clear.cause = 0;
                        /* The CCITT code for DTE-originated    */
      cb_clear.diagnostic = 0;
                        /* No further information               */

      (void)printf("%s: Clearing the call.",argv[0]);

/* The x25_call_clear function can return information from the */
/* clear confirmation packet. However, this isn't required      */
/* here, so set the third parameter to NULL.                    */

      (void)x25_call_clear(conn_id,&cb_clear,(struct cb_msg_struct
                              *)NULL);
    }


/*****************************************************************/
/* If the message received was a clear-indication,              */
/* print out a message before terminating the program.          */

/*****************************************************************/
    else if (cb_msg.msg_type == X25_CLEAR_INDICATION)
    {
    (void)printf("%s: Call cleared. Cause = 0x%02x Diagnostic =
                0x%02x\n",
        argv[0], cb_msg.msg_point.cb_clear->cause,
        cb_msg.msg_point.cb_clear->diagnostic);
    }
```

```
/*****************************************************************/
/* Finally, tidy up by removing the counter and terminating    */
/* the API.                                                    */

/*****************************************************************/
    (void)x25_ctr_remove(ctr_id);
    (void)x25_term(&cb_link_name);
  }
  return(0);
}
```

Index

A

address families, Sockets, 9–7
addresses, binding socket, 9–13
addresses, socket
    obtaining, 9–14
    setting, 9–14
AIX API error codes, HCON programming, 4–42
AIX API errors, HCON programming, 4–14
AIX API file transfer, HCON programming, 4–14
AIX API logical terminal interface, HCON programming, 4–14
AIX API message interface, HCON programming, 4–14
AIX API session control, HCON programming, 4–13
AIX data structures, HCON programming, 4–11
AIX header files, HCON programming, 4–11
AIX interface for HCON API, HCON programming, 4–11
AIX Network Management/6000. *See* SNMP; xgmon
allo_str, 8–5, 8–6
alloc_listen, 8–9
allocating memory with XDR, RPC, 7–26
alphabetic list of NIS Subroutines, 2–3
alphabetical list of DBM subroutines, 2–1
alphabetical list of NDBM subroutines, 2–2
API for X.25
    C subroutines, inclusion in, 10–2
    calls
        clearing, 10–15
        interrupting, 10–16
        receiving fast–select data, 10–15
        rejecting, 10–15
        resetting, 10–16
        terminating, 10–15
    data
        acknowledging, 10–14
        asking for acknowledgement, 10–14
        transferring, 10–14
    data receipt, procedure of, 10–15
    data transfer, long messages procedure, 10–14
    identifiers, use in, 10–2
    shared memory, absence of, 10–5
    subroutines, calling, 10–2
    X.25 communications, use in, 10–1
API program flow, HCON programming, 4–9
application programming interface. *See* API for X.25
application programming interface (API), HCON programming, 4–8
applications for X.25
    counters
        obtaining a, 10–11
        use of, 10–10
    previous releases, changes to, 10–3
    processes, use of, 10–6
    programming for, introduction to, 10–1
arbitrary data types, RPC
    example, 7–61
    passing, 7–25
array, example, XDR, 3–37
array data types, XDR, 3–12
assigning procedure numbers, RPC, 7–21
assigning program numbers, RPC, 7–20
assigning version numbers, RPC, 7–21
attr_str, 8–5, 8–10
authentication
    data encryption standard, RPC, 7–12
    NULL, RPC, 7–11
    RPC, 7–10
    UNIX, RPC, 7–11
authentication protocol, RPC, 7–10
AUTOLOG profile, using, testing, linking, 4–74
AUTOLOG, HCON programming, 4–19
    program examples. *See* AUTOLOG, HCON programming
automatic logon commands, HCON programming, 4–25

B

basic filter primitives, XDR, 3–17
batching, RPC, 7–29
binding names to sockets, concepts. *See* bind subroutine
binding process, RPC, 7–4
block size, XDR, 3–2
boolean data types, XDR, 3–9
booleans, RPC, 7–36
broadcasting
    example, RPC, 7–57
    RPC, 7–30
    server side, RPC, 7–39
byte strings, XDR, 3–13

C

C language syntax for NIDL, NCS, 5–50
C preprocessor, RPC, 7–38

call for X.25
accepting an incoming, subroutines used, 10–14
determining message number, using x25_ctr_test, 10–11
listen identifier, restrictions on, 10–12
making an outgoing, using cb_call_struct subroutine, 10–13
obtaining a listen identifier, using x25_listen, 10–12
receiving an incoming, subroutines used, 10–13
rejecting an incoming, subroutines used, 10–14
removing a listen identifier, using x25_listen, 10–12
using a listen identifier, using x25_listen, 10–12
waiting for an incoming, using x25_ctr_wait subroutine, 10–11
call message, RPC, 7–6
call–back procedures
example, RPC, 7–84
RPC, 7–30
calls for X.25, API level
allocating a PVC, 10–8
connection identifier restrictions, 10–10
connection identifier uses, 10–9—10–10
counter identifier use, 10–10
counters use, 10–10
freeing a PVC, 10–8
initializing, 10–8
obtaining a connection identifier, 10–9
terminating, 10–8
terminating each X.25 port, 10–8
canonical standard, XDR, 3–1
changing time outs, RPC, 7–39
closing sockets. *See* socket shutdown
Command Line Manager, SNMP, 6–22
communications for X.25, standard, origin of, 10–1
compiling programs, RPC, 7–27
compiling, HCON programming
AIX API programs, C, FORTRAN, Pascal, 4–69
file transfer program, C, FORTRAN, Pascal, 4–68
host API programs, C, FORTRAN, Pascal, 4–70
confirm_str, 8–5, 8–11
connect sockets, 9–15
connected sockets, create a pair, 9–15
connection identifier for X.25, subroutines, use in, 10–9
connectionless sockets, 9–16
connections, socket. *See* connect subroutine
constants
luxsna.h constants, 8–28
error codes, 8–28
request codes, 8–28, 8–30
status codes, 8–28
RPC, 7–34
XDR, 3–15
constructed filter primitives, XDR, 3–18
converting local procedures
example, RPC, 7–75
RPC, 7–38
counter for X.25
removing, 10–12
restrictions on, 10–12
counter identifier for X.25, subroutines
x25_call, 10–11
x25_call_accept, 10–11
x25_ctr_test, 10–11
x25_ctr_wait, 10–11
x25_listen, 10–11
x25_pvc_alloc, 10–11
counter subroutines for X.25, list of, 10–3
cp_str, 8–5, 8–11

D

data description, example, XDR, 3–40
data encryption standard authentication
Diffie–Hellman encryption, RPC, 7–15
example, RPC, 7–53
protocol, RPC, 7–14
RPC, 7–12
data link control, generic, 1–1
data link control, IEEE 802.3 ethernet, 1–26
data link control, qualified logical link control, 1–52
data link control, standard ethernet, 1–35
data link control, synchronous, 1–44
data link control, token–ring, 1–16
data packets for X.25, receipt of, 10–15
data stream, XDR
creating, 3–20
destroying, 3–22
implementing, 3–21
manipulating, 3–21
using, 3–20
data transfer, socket, 9–18
data types
array, XDR, 3–12
boolean, XDR, 3–9
enumeration, XDR, 3–9
floating–point, XDR, 3–9
integer, XDR, 3–8
opaque, XDR, 3–11
XDR, 3–8
datagram services, connectionless, 9–16
DBM, alphabetical list of subroutines, 2–1
deal_str, 8–5, 8–13
declarations
RPC, 7–35
XDR, 3–5
definitions, RPC, 7–31
DES. *See* data encryption standard
/dev/x25sn special file, 10–9
Diffie–Hellman encryption, RPC, 7–15

discriminated union
example, XDR, 3–42
XDR, 3–14
dlc802.3, IEEE ethernet data link control, 1–26
dlcether, standard ethernet data link control, 1–35
DLCQLLC, qualified logical link control, 1–52
dlcsdlc, synchronous data link control, 1–44
dlctoken, token–ring data link control, 1–16
domain name resolution, Sockets, 9–24
domain name translation, Sockets, 9–24

E

enhancements, planned, XDR, 3–2
enumeration data type, XDR, 3–9
enumerations
RPC, 7–33
XDR, 3–5
erro_str, 8–5, 8–14
error codes, 8–28
error codes for X.25
errno, use of, 10–6
non–X.25 specific, list of, 10–20
x25_errno, use of, 10–6
ethernet data link control, IEEE 802.3, 1–26
example
array, XDR, 3–37
broadcasting, RPC, 7–57
call–back procedures, RPC, 7–84
converting local procedures, RPC, 7–75
data description, XDR, 3–40
DES authentication, RPC, 7–53
discriminated union, XDR, 3–42
generating XDR routines, RPC, 7–80
highest layer, RPC, 7–58
intermediate layer, RPC, 7–59
justification for using, XDR, 3–32
local to remote, RPC, 7–75
lowest layer, RPC, 7–63
multiple program versions, RPC, 7–73
passing arbitrary data types, RPC, 7–61
passing linked lists, XDR, 3–29
ping program, RPC, 7–56
pointers, XDR, 3–35
rcp on TCP, RPC, 7–69
select subroutine, RPC, 7–68
UNIX authentication, RPC, 7–50
using an array, XDR, 3–37
using XDR, 3–36
example code, demonstration purposes, use only as, 10–21
example programs
compiling, 10–21
preparing, 10–21
running, 10–21
example programs for X.25
PVC
receiving data using, 10–23—10–26
sending data using, 10–28—10–31
pvcrcv, 10–23—10–26
pvcxmit, 10–28—10–31
SVC
making a call using, 10–39—10–44
receiving a call using, 10–33—10–37
svcrcv, 10–33—10–37
svcxmit, 10–39—10–44
example programs, Sockets, understanding, 9–26
exceptions, RPCL rules, 7–36
explicit logon, HCON programming, 4–17
ext_io_str, 8–5, 8–15
eXternal Data Representation (XDR)
See also XDR
allocating memory with RPC, 7–26
array, example, 3–37
array data types, 3–12
basic filter primitives, 3–17
block size, 3–2
boolean data types, 3–9
byte strings, 3–13
canonical standard, 3–1
constants, 3–15
constructed filter primitives, 3–18
data description, example, 3–40
data stream
creating, 3–20
destroying, 3–22
implementing, 3–21
manipulating, 3–21
using, 3–20
data types, 3–8
array, 3–12
boolean, 3–9
constants, 3–15
discriminated unions, 3–14
enumerations, 3–9
floating–point, 3–9
integer, 3–8
opaque, 3–11
optional data, 3–16
strings, 3–13
structures, 3–13
type definitions, 3–15
union, optional data, 3–16
voids, 3–15
declarations, 3–5
discriminated union, 3–14
example, 3–42
enhancements, planned, 3–2
enumeration data type, 3–9
enumerations, 3–5
filter primitives, 3–17
basic, 3–17
constructed, 3–18
floating–point data types, 3–9
generating routines with RPC, 7–38
example, 7–80
integer data types, 3–8

justification for using, example, 3–32
language
declarations, 3–5
enumerations, 3–5
lexical notes, 3–5
specifications, 3–5
structure, 3–5
syntax notes, 3–7
unions, 3–5
lexical notes, 3–5
library, 3–4
library filter primitives, 3–17
linked lists, example, 3–29
memory allocation in RPC, 7–26
non–filter primitives, 3–20
opaque data types, 3–11
operation directions, 3–4
optional data, 3–16
overview, 3–1
passing linked lists, example, 3–29
pointers, example, 3–35
primitives
filter, 3–17
non–filter, 3–20
RPC, using with, 3–4
strings, 3–13
structures, 3–5, 3–13
subroutine format, 3–3
syntax notes, 3–7
type definitions, 3–15
unions, 3–5
discriminated unions, 3–14
optional data, 3–16
using, example, 3–36
using an array, example, 3–37
voids, 3–15

F

features, RPC, 7–29
file transfer
prerequisite hardware, 8–55
prerequisite software, 8–55
starting, 8–55
file transfer error codes, HCON programming, 4–28
file transfer programming interface, HCON programming
asynchronous, synchronous file transfer, 4–4
cfxfer, fxfer, 4–4
data structures, 4–5
header files, 4–5
security, 4–4
file transfer, HCON programming, AIX API, 4–14
files, special, 8–5
luxsna.h header file, 8–5
filter primitives
basic, XDR, 3–17
constructed, XDR, 3–18
XDR, 3–17
flags for X.25, use of, 10–5
floating–point data types, XDR, 3–9
flush_str, 8–5, 8–20
fmh_str, 8–5, 8–21
fxc C data structure, HCON programming. *See* file transfer programming interface
fxc Pascal declarations, HCON programming. *See* file transfer programming interface

G

gdlc, generic data link control, 1–1
generating XDR routines
example, RPC, 7–80
RPC, 7–38
generic data link control, 1–1
get_parms, 8–22
gstat_str, 8–5, 8–22

H

HCON File Transfer Program Interface, compiling, 4–68
HCON host API programs, compiling, 4–70
HCON programming
AIX API C language structures, 4–12
AIX API errors, 4–14
AIX data structures, 4–11
AIX interface for HCON API, 4–11
API program flow, 4–9
application programming interface (API), 4–8
AUTOLOG, 4–19
automatic logon commands, 4–25
compiling programs, 4–27
example programs, 4–25
explict, implicit logon, 4–17
file transfer programming data structures, C, Pascal, 4–5
file transfer programming interface, 4–4
FORTRAN API language structures, 4–13
host interface errors, 4–16
host interface for HCON API, 4–15
LAF language, 4–21
LAF script statements, 4–24
Logon Assist Feature (LAF), 4–20
Pascal language API structures, 4–12
HCON programming error codes
AIX API, 4–42
file transfer, 4–28
host API, 4–50
HCON programming examples, 4–25
API, 4–26
file transfer program interface, 4–25
header file, luxsna.h, 8–5
header files, HCON programming
AIX, g32const.inc, g32hfile.inc, g32keys.inc, g32types.inc, g32_api, g32_keys_h, 4–11
file transfer, fxfer.h, fxconst.inc, fxfer.inc, fxhfile.inc, 4–5
header files, Sockets, in.h, nameser.h, netdb.h, resolv.h, socket.h, socketvar.h, un.h, 9–5

highest layer, example, RPC, 7–58
host API, HCON programming
  installing on MVS/TSO systems, 4–64
  installing on VM/CMS systems, 4–66
host error codes, HCON programming, 4–50
host file flags, HCON programming, 4–7
host interface errors, HCON programming, 4–16
host interface for HCON API, HCON programming, 4–15
host message interface, HCON programming, 4–16
host name translation, Sockets, 9–22
host session control, HCON programming, 4–15

I

I/O modes, Sockets, 9–20
IEEE 802.3 ethernet data link control, 1–26
implicit logon, HCON programming, 4–17
include files, Sockets. *See* header files, Sockets
inetd daemon, starting RPC, 7–27
initialization subroutines for X.25
  changes to, 10–4
  list of, 10–3
installing MVS/TSO host API, HCON programming, 4–64
installing VM/CMS host API, HCON programming, 4–66
integer data types, XDR, 3–8
intermediate layer
  example, RPC, 7–59
  RPC, 7–23
Internet address translation, Sockets, 9–23
ioctls, socket, SIOCGPGRP, SIOCSPGRP, FIOASYNC. *See* out–of–band data, Sockets

J

justification for using, example, XDR, 3–32

L

LAF language, HCON programming, 4–21
LAF script statements, HCON programming, 4–24
LAF script, HCON programming, using, 4–71
language, RPC, 7–31
language descriptions, RPC, 7–31
language specifications, XDR, 3–5
lb_$ library routines, NCS, 5–89
lexical notes, XDR, 3–5
library, XDR, 3–4
library filter primities, XDR, 3–17
library routines, NCS
  lb_$, 5–89
  pfm_$, 5–86
  rpc_$, 5–81
  uuid_$, 5–94
linked lists, example, XDR, 3–29
linking programs, RPC, 7–27
listen identifier for X.25, purpose of, 10–12
local procedures, example converting to remote, RPC, 7–75
Location Broker daemons, NCS
  llbd daemon, 5–75, 5–78
  nrglbd daemon, 5–75, 5–79
Location Broker, NCS, 5–3, 5–75
  Client Agent, 5–3, 5–75, 5–77
  lb_$ library routines, 5–89
logical path, HCON programming, 4–3
logical terminal interface, HCON programming, AIX API, 4–14
Logon Assist Feature (LAF), HCON programming, 4–20
  See also Logon Assist Feature, HCON programming
Logon Assist Feature script, HCON programming, using sample, testing, 4–71
lowest layer, example, RPC, 7–63
luxsna.h, 8–5
  constants, 8–28
    error codes, 8–28
    request codes, 8–28, 8–30
    status codes, 8–28
  structures, 8–5
    allo_str, 8–5, 8–6
    alloc_listen, 8–9
    attr_str, 8–5, 8–10
    confirm_str, 8–5, 8–11
    cp_str, 8–5, 8–11
    deal_str, 8–5, 8–13
    erro_str, 8–5, 8–14
    ext_io_str, 8–5, 8–15
    flush_str, 8–5, 8–20
    fmh_str, 8–5, 8–21
    get_parms, 8–22
    gstat_str, 8–5, 8–22
    pip_str, 8–5, 8–24
    prep_str, 8–5, 8–24
    read_out, 8–5, 8–25
    stat_str, 8–5, 8–27
    write_out, 8–5, 8–27

M

management subroutines for X.25
  list of, 10–3
  security permissions, requirements for, 10–7
marking records, RPC, messages, 7–9
memory allocation with XDR, RPC, 7–26
message interface, HCON programming
  AIX API, 4–14
  host API, 4–16
message protocol, RPC, 7–5
  requirements, 7–5
message structure, RPC, 7–5
MIB, 6–3
  using database, 6–6
  variables
    SNMP daemon support for, 6–16
    terminology, 6–6
    working with, 6–9

model, RPC, 7–2
multiple program versions, example, RPC, 7–73
MVS/TSO, HCON programming, installing host API, 4–64

N

name, socket's bound, 9–13
NCS
  commands, 5–80
  daemons, 5–80
    See also Location Broker daemons
  introduction, 5–1
  library routines
    lb_$, 5–89
    pfm_$, 5–86
    rpc_$, 5–81
    uuid_$, 5–94
  Location Broker, 5–75
  NIDL, 5–3, 5–16
    See also NIDL; NIDL compiler
  Remote Procedure Call (RPC) runtime library. *See* RPC runtime library
  remote procedure calls, 5–6
  RPC runtime library, 5–3, 5–4
    client routines, 5–14
    conversion routines, 5–15
    server routines, 5–14
  UUIDs, 5–4, 5–24
  writing applications, 5–24
NDBM, alphabetical list of subroutines, 2–2
network address translation, Sockets, 9–22
network byte order, Sockets, translation, 9–23
Network Computing System. *See* NCS
network host name, Sockets, translation, 9–24
Network Interface Definition Language, NCS. *See* NIDL
network name translation, Sockets, 9–22
NIDL compiler, NCS, 5–3, 5–30
  concepts, 5–16
NIDL, NCS
  applications
    building of, 5–49
    writing of, 5–24
  banking example, 5–17
  binop example, 5–18
  client programs, writing of, 5–43
  concepts, 5–16
  handles, use of, 5–21
  handles and bindings, managing of, 5–34
  interface definitions
    compiling of, 5–30
    writing of, 5–25
  server program, writing of, 5–46
  stub functions, 5–20
  using C syntax, 5–50
  using Pascal syntax, 5–62
  with FORTRAN, 5–73
  writing manager procedures, 5–48
  writing server main procedure, 5–46
NIS, alphabetic list of subroutines, 2–3
non–filter primitives, XDR, 3–20
NULL authentication, RPC, 7–11

O

opaque data, RPC, 7–37
opaque data types, XDR, 3–11
operation directions, XDR, 3–4
optional data, XDR, 3–16
options, socket
  getting. *See* getsockopt subroutine
  setting. *See* setsockopt subroutine
out–of–band data, Sockets, 9–19

P

Pascal language syntax for NIDL, NCS, 5–62
passing arbitrary data types, example, RPC, 7–61
passing linked lists, example, XDR, 3–29
pfm_$ library routines, NCS, 5–86
physical path, HCON programming, 4–3
ping program, example, RPC, 7–56
pip_str, 8–5, 8–24
pointers, example, XDR, 3–35
port mapper, RPC, registering ports, 7–17
port mapper procedures, RPC, 7–19
port mapper program, RPC, 7–17
port mapper protocol, RPC, 7–18
ports, registering, RPC, 7–17
prep_str, 8–5, 8–24
presentation space, HCON programming, 4–3
primitives
  filter, XDR, 3–17
  non–filter, XDR, 3–20
procedure numbers, RPC, assigning, 7–21
program numbers, RPC, assigning, 7–20
programming, RPC, 7–20
programming in RPC, 7–20
programs, RPC, 7–34
protocol, RPC message, 7–5
protocol compiler, rpcgen, RPC, 7–37
protocol name translation, Sockets, 9–23
protocols, socket, families, PF_INET, PF_UNIX, 9–11

Q

qualified logical link control, 1–52

R

rcp on TCP, example, RPC, 7–69
read_str, 8–5, 8–25
record marking, RPC, messages, 7–9
registered programs, RPC, 7–21
registered RPC programs, 7–21
registering ports, RPC, 7–17

Remote Procedure Call (RPC)
- allocating memory with XDR, 7–26
- arbitrary data types
  - example, 7–61
  - passing, 7–25
- assigning procedure numbers, 7–21
- assigning program numbers, 7–20
- assigning version numbers, 7–21
- authentication, 7–10
  - data encryption standard, 7–12
  - NULL, 7–11
  - UNIX, 7–11
- authentication protocol, 7–10
- batching, 7–29
- binding process, 7–4
- booleans, 7–36
- broadcasting, 7–30
  - example, 7–57
  - server side, 7–39
- C preprocessor, 7–38
- call message, 7–6
- call–back procedures, 7–30
  - example, 7–84
- changing time outs, 7–39
- compiling programs, 7–27
- constants, 7–34
- converting local precedures, 7–38
- converting local procedures, example, 7–75
- data encryption standard authentication, 7–12
  - example, 7–53
  - protocol, 7–14
- declarations, 7–35
- definitions, 7–31
- DES authentication, example, 7–53
- Diffie–Hellman encryption, 7–15
- enumerations, 7–33
- exceptions to language rules, 7–36
- features, 7–29
- generating XDR routines, 7–38
  - example, 7–80
- highest layer, example, 7–58
- inetd daemon, starting from, 7–27
- intermediate layer, 7–23
  - example, 7–59
- language, 7–31
  - constants, 7–34
  - declarations, 7–35
  - definitions, 7–31
  - descriptions, 7–31
  - enumerations, 7–33
  - example of ping program, 7–56
  - exceptions to rules, 7–36
    - booleans, 7–36
    - opaque data, 7–37
    - strings, 7–36
    - voids, 7–37
  - programs, 7–34
  - rpcgen protocol compiler, 7–37
  - structures, 7–32
  - syntax requirements, 7–36
  - type definitions, 7–33
  - unions, 7–32
- linking programs, 7–27
- local procedures, example converting to remote, 7–75
- lowest layer, example, 7–63
- marking records in messages, 7–9
- memory allocation with XDR, 7–26
- message protocol, 7–5
  - requirements, 7–5
- messages, 7–5
  - call, 7–6
  - marking records, 7–9
  - protocol requirements, 7–5
  - reply, 7–7
  - structure, 7–5
- model, 7–2
- multiple program versions, example, 7–73
- NULL authentication, 7–11
- opaque data, 7–37
- overview, 7–1
- port mapper, registering ports, 7–17
- port mapper procedures, 7–19
- port mapper program, 7–17
- port mapper protocol, 7–18
- procedure numbers, assigning, 7–21
- program numbers, assigning, 7–20
- programming, 7–20
- programs, 7–34
- rcp on TCP, example, 7–69
- record marking in messages, 7–9
- records, marking in messages, 7–9
- registered programs, 7–21
- registering ports, 7–17
- reply message, 7–7
- rpcgen protocol compiler
  - broadcast on the server side, 7–39
  - C preprocessor, 7–38
  - changing time outs, 7–39
  - converting local to remote, 7–38
  - generating XDR routines, 7–38
  - other information passed to server, 7–39
  - using, 7–37
- select subroutine
  - example, 7–68
  - server side, 7–30
- semantics, 7–3
- server procedures, 7–39
- servers
  - broadcasting, 7–39
  - select subroutine, 7–30
- strings, 7–36
- structures, 7–32
- time outs, changing, 7–39
- transports, 7–3
- type definitions, 7–33
- understanding model, 7–2
- unions, 7–32

UNIX authentication, 7–11
example, 7–50
using rcp on TCP, 7–69
version numbers, assigning, 7–21
voids, 7–37
XDR, using with, 3–4
Remote Procedure Call runtime library, NCS. *See* RPC runtime library
remote procedure calls, NCS, paradigm for, 5–6
remote procedures, convert local procedures to, RPC, 7–38
reply message, RPC, 7–7
request codes, 8–28
reset for X.25, handling of, 10–16
resolver, Sockets, resolver subroutines, 9–24
RPC. *See* Remote Procedure Call (RPC)
RPC runtime library, NCS, 5–3, 5–4, 5–13
client routines, 5–14
conversion routines, 5–15
routines, 5–13
server routines, 5–14
rpc_$ library routines, NCS, 5–81
rpcgen, protocol compiler, RPC, 7–37
RPCL
exceptions to the rules, 7–36
ping program, example, 7–56
syntax requirements for program definition, 7–36

S

select subroutine
example, RPC, 7–68
server side, RPC, 7–30
semantics, RPC, 7–3
server connections, Sockets, 9–16
server procedures, RPC, 7–39
servers
broadcasting, RPC, 7–39
select subroutine, RPC, 7–30
service name translation, Sockets, 9–23
session control, HCON programming
AIX API, 4–13
host API, 4–15
Session Modes, HCON programming, 4–1
session, HCON programming, 4–3
shutdown sockets, 9–21
Simple Network Management Protocol. *See* SNMP
SNA Services/6000
developing special SNA functions, 8–32
configurations, 8–34
functional characteristics, 8–34
error codes, 8–28
example programs
file transfer
rcvfrom.c, 8–56
sendto.c, 8–56
transaction program
local, 8–46
mapped, 8–51
mapped remote, 8–52
remote, 8–48
extended interface, using, 8–44
file transfer, 8–55
example program
rcvfrom.c, 8–56
sendto.c, 8–56
limited interface, using, 8–44
local transaction program, example program, 8–46
LU0 facility, 8–37
mapped remote transaction program, example program, 8–52
mapped transaction program, example program, 8–51
parameter passing, 8–44
programs
rcvfrom.c, 8–56
sendto.c, 8–56
rcvfrom.c, file transfer, example program, 8–56
request codes, 8–30
sendto.c, file transfer, example program, 8–56
signals, 8–45
transaction program
See also remote transaction program name (RTPN)
extended interface, 8–44
limited interface, 8–44
local, example program, 8–46
mapped, example program, 8–51
mapped remote, example program, 8–52
remote, example program, 8–48
signals, 8–45
writing, 8–44
guidelines, 8–44
transferring files, 8–55
writing, transaction programs, 8–44
guidelines, 8–44
Writing Generic Programs for, 8–57
snaopen, 8–55
SNMP, 6–3
agent, function of, 6–8
API subroutine library, 6–10
list of subroutines, 6–12
Command Line Manager, 6–22
daemon, 6–13
configuring, 6–13
implementation restrictions, 6–21
processing, 6–14
RFC conformance, 6–20
support for SET request processing, 6–17
support for the EGP family of MIB variables, 6–16

formatting VGM windows, xgmon, 6–25
internal database, VGM, xgmon, 6–25
intrinsic functions, xgmon
alphabetic list, 6–37
extending of, 6–23
functional list, 6–39
how to create, 6–43
library commands, creating of, 6–24
library commands, xgmon
how to create, 6–46
how to modify, 6–48
Management Information Base. *See* MIB
monitor, function of, 6–8
programming VGMs, xgmon, 6–24
VGM run–time environment, 6–30
working with VGM data types, 6–29
working with VGM variables, 6–26
xgmon library programs, structure of, 6–30
xgmon Overview for Programmers, 6–1
xgmon programming utility, 6–23
conditional statements, 6–33
expressions, 6–34
intrinsic functions, 6–35
iteration statements, 6–33
operators, 6–34
simple statements, 6–32
snmpd daemon. *See* SNMP, daemon
sockaddr structure, Sockets, sa_family, 9–6
socket
address storage, 9–8
blocking, non–blocking mode, 9–20
data transfer, 9–18
discarding. *See* close subroutine
ioctls, 9–18
network address translation, 9–22
out–of–band data, 9–19
protocols, 9–9
types, 9–9
socket address, data structures, 9–6
socket addresses, TCP/IP, 9–9
socket I/O modes, 9–20
socket interface to networks, 9–4
socket layer, 9–3
socket options, get, set, 9–17
socket subroutine library, 9–5
Sockets, 9–1
sockets interface, 9–3
Special Files, /dev/x25sn, 10–9
special files, 8–5
standard ethernet data link control, 1–35
starting from the inetd daemon, RPC, 7–27
stat_str, 8–5, 8–27
status codes, 8–28
strings
RPC, 7–36
XDR, 3–13
structures
luxsna.h structures, 8–5
allo_str, 8–5, 8–6
alloc_listen, 8–9
attr_str, 8–5, 8–10
confirm_str, 8–5, 8–11
cp_str, 8–5, 8–11
deal_str, 8–5, 8–13
erro_str, 8–5, 8–14
ext_io_str, 8–5, 8–15
flush_str, 8–5, 8–20
fmh_str, 8–5, 8–21
get_parms, 8–22
gstat_str, 8–5, 8–22
pip_str, 8–5, 8–24
prep_str, 8–5, 8–24
read_out, 8–5, 8–25
stat_str, 8–5, 8–27
write_out, 8–5, 8–27
RPC, 7–32
XDR, 3–5, 3–13
subroutine format, XDR, 3–3
subroutines for X.25
new, list of, 10–4
obsolete, list of, 10–4
synchronous data link control, 1–44
syntax notes, XDR, 3–7
syntax requirements, program definition, RPCL, 7–36

T

TCP/IP socket addresses, 9–9
termination subroutines for X.25
changes to, 10–4
list of, 10–3
time outs, changing, RPC, 7–39
token–ring data link control, 1–16
transaction program, 8–44
transports, RPC, 7–3
troubleshoot file transfer program interface errors, HCON programming, 4–28
troubleshoot HCON API errors, HCON programming, 4–42
troubleshoot host API errors, HCON programming, 4–50
TSO, HCON programming, running example programs, 4–26
type definitions
RPC, 7–33
XDR, 3–15
types, socket, SOCK_DGRAM, SOCK_STREAM, SOCK_RAW, 9–10

U

unions
- discriminated, XDR, 3–14
- optional data, XDR, 3–16
- RPC, 7–32
- XDR, 3–5

Unique Universal Identifiers. *See* UUIDs
UNIX authentication
- example, RPC, 7–50
- RPC, 7–11

using a Logon Assist Feature script, HCON programming, 4–71
using an array, example, XDR, 3–37
using an AUTOLOG profile, 4–74
using rcp on TCP, RPC, 7–69
using XDR, example, 3–36
/usr/include/luxsna.h, 8–5
utilities, HCON programming, automatic logon commands, 4–25
uuid_$ library routines, NCS, 5–94
UUIDs, NCS, 5–4, 5–24

V

version numbers, RPC, assigning, 7–21
VGM. *See* xgmon, virtual G machines
VM/CMS, HCON programming, installing host API, 4–66
voids
- RPC, 7–37
- XDR, 3–15

W

write_out, 8–5, 8–27

X

X.25 adapters, multiple, support for, 10–4
X.25 Device Handler, special file, 10–9
X.25 management subroutines, changes to, 10–5
X.25 network
- example of, 10–1
- subroutines, list of, 10–3

X.25 network subroutines, changes to, 10–4
X.25 protocol, future developments, allowance for, 10–5
X.25 structures, use of, 10–5
XDR. *See* eXternal Data Representation (XDR)
xgmon
- intrinsic functions
  - alphabetic list, 6–37
  - extending of, 6–23
  - functional list, 6–39
  - how to create, 6–43
- library commands
  - creating of, 6–24
  - how to create, 6–46
  - how to modify, 6–48
- library programs, structure of, 6–30
- overview for programmers, 6–1
- programming utility, 6–23
  - conditional statements, 6–33
  - expressions, 6–34
  - intrinsic functions, 6–35
  - iteration statements, 6–33
  - operators, 6–34
  - simple statements, 6–32
- virtual G machines (VGMs)
  - data types, 6–29
  - formatting VGM windows, 6–25
  - internal database, 6–25
  - programming of, 6–24
  - run–time environment, 6–30
  - variables, 6–26

Reader's Comment Form

AIX Communications Programming Concepts for IBM RISC System/6000

SC23–2206–00

Please use this form only to identify publication errors or to request changes in publications. Your comments assist us in improving our publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your IBM-approved remarketer. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

☐ If you would like a reply, check this box. Be sure to print your name and address below.

Page	Comments

Please contact your IBM representative or your IBM–approved remarketer to request additional publications.

Please print

Date ____________

Your Name ______________________________

Company Name ______________________________

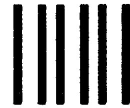
Mailing Address ______________________________

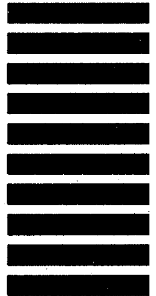
____

____

Phone No. (   ) ____________________
Area Code

No postage necessary if mailed in the U.S.A





BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 997
11400 Burnet Rd.
Austin, Texas 78758–3493

Fold

Cut or Fold Along Line

Fold and Tape

Please Do Not Staple

Tape

© IBM Corp. 1990

International Business Machines
Corporation
11400 Burnet Road
Austin, Texas 78758–3493

Printed in the
United States of America
All Rights Reserved

SC23-2206-00